AFIT/GCS/ENG/93D-05

A PARALLEL COMPUTATIONAL FLUID DYNAMICS UNSTRUCTURED GRID GENERATOR

THESIS

Deborah E. Davis, Captain, USAF

AFIT/GCS/ENG/93D-05

AFIT/GCS/ENG/93D-05

A PARALLEL COMPUTATIONAL FLUID DYNAMICS UNSTRUCTURED GRID GENERATOR

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

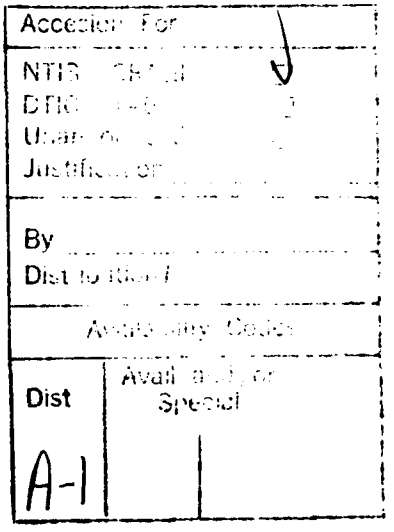
Requirements for the Degree of

Master of Science in Computer Engineering

Deborah E. Davis, B.S.

Captain, USAF

December 1993



Preface

The purpose of this research was to develop a parallel computational fluid dynamics (CFD) unstructured grid generator using Delaunay triangulation techniques. This topic is of interest to AFIT, NASP, Wright Labs, and the CFD community in general. Researchers have done little work in this area, but as the number of parallel CFD flow solvers increases, the grid generation process will become a bottleneck.

In order to accomplish this goal, I had a lot to learn about both CFD and parallel processing. I had a great deal of help in this effort. I would especially like to thank my advisor, LtCol Hobart, and my committee members, Capt Doty, Dr Lamont, and Dr Beran for their help in trying to get a grasp on all the concepts involved. Without their expertise I would have been lost. A special thanks, as well, to Capt Frank Smith for his patient explanations and access to his sequential code.

During the development phase of my thesis I received a great deal of help from several members of the parallel processing community. I would like to thank the members of the Mathematics Sciences Section at Oak Ridge [illegible]ratory, especially Barry Peyton and Dave MacKay for their help in providing me with their parallel recursive spectral bisection code. I would also like to thank Trey Arthur and Michael Bockelie of Computer Sciences Corporation for their help in getting a copy of the parallel advancing-front unstructured grid generator (VGRIDSG) they developed for NASA Langley.

Finally, I would like to thank my family and friends for their continual support whether they were close or far away. A special thanks goes to my husband, Jon, for his support and understanding, as well as his technical help when it came to aeronautical engineering, math, and Fortran problems.

Deborah E. Davis

Table of Contents

List of Figures

List of Tables

Abstract

This research addressed the development of a parallel computational fluid dynamics unstructured grid generator using Delaunay triangulation. The generator is applied to simple elliptical and cylindrical two-dimensional bodies. The methodologies used in the development of the generator included Watson's point insertion algorithm, Holmes and Snyder's point creation algorithm, a discretized surface definition, Anderson's clustering function, and a Laplacian smoother. The first version of the software involved a processor boundary exchange at the end of each iteration with no inter-processor communications during the iterations. The second version used inter-processor communication during each iteration instead of the boundary exchange. Version 1 proved to be unscalable due to the interdependency of the triangular elements. The code did scale to two processors, in some cases, and for these cases portions of the code demonstrated a speedup of 1.8. Version 2 could be used on multiple processors, but did not provide continued speedup past two processors due to the communication overhead. Two distribution methodologies, a simple 360-degree distribution and recursive spectral bisection (RSB), were examined. For the initial grid distribution, the distribution generated by the RSB code would be similar to the distribution generated by the 360-degree methodology and would require significantly more time to execute. The advantages associated with the RSB distribution methodology were not apparent for the initial distribution.

A PARALLEL COMPUTATIONAL FLUID DYNAMICS UNSTRUCTURED GRID GENERATOR

Chapter 1: Thesis Background

Introduction

Computational Fluid Dynamics (CFD) is a numerical representation of a fluid flow around or through a solid body. This body can be an aircraft, car, missile, or any other vehicle where the interaction with the surrounding flowfield is an important consideration. The design of such vehicles is often based on CFD methods. The accuracy of the results of such calculations is limited by the capabilities of the computer on which the calculations are performed. Extended periods of time can be required to complete numerically intensive CFD computations. One way to improve CFD computational performance, and thus increase the accuracy possible, is to use parallel computers. One part of the parallel CFD process that has not been studied extensively is grid generation. That part of the CFD process is the focus of this thesis. This chapter first presents a background of the problem including why the topic is important and the concepts that are important for a discussion of CFD and parallel computing. Following the background information, this chapter provides the assumptions that apply to the thesis research and how these assumptions limit the scope of the topic. Then, the chapter presents the approach required to solve the thesis problem.

Background

Previously, technological advances provided sequential computers that decreased the time required to perform CFD computations. Current sequential computers are reaching

maximum speeds. One of the most promising solutions to this barrier is the use of parallel computers. This option has led researchers to work on parallelizing CFD codes. A basic background in CFD and parallel computing concepts is required to fully understand a problem description dealing with the parallelization of a CFD application. Before these concepts are presented, this section discusses the state of parallel CFD research at the Air Force Institute of Technology (AFIT).

AFIT Parallel CFD Research. The Air Force Institute of Technology is currently working toward transitioning much of its CFD research from vector to massively parallel supercomputers. The effort is a five-year undertaking, of which this year is the first. Transitioning to parallel architectures will allow AFIT to stay up to date with current technology and provide a better platform for CFD research.

The AFIT Aeronautics and Astronautics department has studied the use of a two-dimensional unstructured grid with a finite-volume solver for Laplace's equation on elliptical objects (Smith, 1992). This research applied to low-speed flow and reconfigurable objects. The department researchers used the Delaunay triangulation method to create the unstructured grid. Their algorithm, hosted on a sequential architecture, combined Watson's and Bowyer's algorithms. The algorithm was an incremental insertion algorithm that sequentially inserted points into the current Delaunay triangulation.

The National AeroSpace Plane (NASP) Joint Program Office is also interested in the development of CFD algorithms for parallel architectures. They are supporting AFIT's effort to transition CFD research. AFIT plans to direct the parallel CFD research toward applications that are of potential use to NASP.

Parallel Computing Concepts. The concepts involved in parallel computing include basic computer concepts, the parallel environment, parallel algorithms, and parallel metrics. Basic computer concepts are not discussed here. Only the unique aspects of the field of parallel computing are presented.

A parallel environment includes the programming language, support software, and architecture. Generally, the languages and support software that are currently available are dependent on the hardware platform. A variety of massively parallel computer architectures are available. One classification system is Flynn's taxonomy (Wilson, 1993:57). From Flynn's taxonomy, the two most common parallel architecture classifications are Multiple Instruction Multiple Data (MIMD) and Single Instruction Multiple Data (SIMD). MIMD architectures provide complete separation of the processors. Each processor controls its own memory and runs asynchronously. Communication between processors is accomplished via message-passing constructs. In a SIMD architecture, the processors share memory. The shared memory eliminates the need for message passing constructs. The processors in a SIMD architecture run synchronously. The same instruction is performed on each processor for different data values simultaneously. Both MIMD and SIMD categories include several more specific architectures. MIMD architectures include hypercubes, meshes, and clusters. SIMD architectures include vector/array processors, and the early Thinking Machines. The new CM-5, by Thinking Machines, is an example of the combination of aspects of both architectures. Ongoing research is focusing on this type of architecture combination using the advantages of both architecture classes.

A parallel algorithm is often effective and efficient only on a specific target architecture which must be carefully considered during the algorithm development. Researchers use two primary metrics to measure the performance of parallel algorithms. The first metric is speedup. Speedup indicates how much faster an application runs on p parallel processors than on one processor. The primary equation for speedup is

$$S_p = T_1 \; / \; T_p \qquad (1)$$

where

S_p = *speedup for p processors*

T_1 = *time the application takes on one processor*

T_p = *time the application takes on p processors*

(Lewis and El-Rewini, 1992: 31). Amdahl expands on this primary equation to limit the speedup possible based on the portion of the code that is sequential. For Amdahl's law, β is the portion of the code that must be computed sequentially. The time required for the sequential portion of the code is βT_1 and the time required for the parallel portion is $(1-\beta)T_1/p$. Amdahl's law expands the primary equation for speed up to

$$S_p = \frac{1}{\beta + \frac{(1-\beta)}{p}} = \frac{p}{\beta * p + (1-\beta)} \quad (2)$$

(Lewis and El-Rewini, 1992: 31-32). Gustafson and Barsis' alternative to Amdahl's law is more optimistic. They assumed that $T_1 = \beta + (1-\beta)p$ and $T_p = \beta + (1-\beta) = 1$. These assumptions resulted in the following form of the speedup equation:

$$S_p = p - (p-1) * \beta \quad (3)$$

This equation is known as the Gustafson-Barsis law (Lewis and El-Rewini, 1992: 32-33).

For an algorithm running on 200 processors, a speedup of 200 is linear speedup. Linear speedup is the ideal speedup for a parallel algorithm. Often, if speedup is not realized for an algorithm, the communication time is dominating the total time and little time is spent on computations.

The second metric is efficiency. Efficiency indicates how efficiently the processors on a parallel machine are used. The equation for efficiency is

$$E_p = S_p / p \quad (4)$$

where

E_p = *efficiency of p processors*

S_p = *speedup for p processors*

p = *number of processors*

(DeCegama, 1989: 7). An efficiency of 1.0, or 100%, indicates that every processor is being used to the full extent of its capabilities. Usually, efficiency measures that are significantly lower than 100% are due to communication time or the unbalanced distribution of the problem over the processors involved.

Several other parallel computing concepts are also important in the development of parallel algorithms. One such concept is data (or domain) versus control decomposition. "In domain decomposition, the domain of the input data are partitioned and the partitions are assigned to different processors. In control decomposition, program tasks are divided and distributed among processors" (Lewis and El-Rewini, 1992: 138). This decomposition is balanced if the amount of work assigned to each processor is equal. The attempt to balance the decomposition is known as "load balancing" (Lewis and El-Rewini, 1992: 137). If an increase in the size of the application can be countered by a corresponding increase in the number of processors used, and the time required for the application remains constant, the application is scalable.

The granularity of an application indicates the amount of processing that can be completed between required message passing events. A "fine-grained" application has few operations between message passing events. A "course-grained" application has numerous operations to perform between message passing events. If the grain is too small, communications can dominate the time required to complete the application. If it is too small, several possibly parallel portions of the code may be executing sequentially, reducing the parallelism of the application (Lewis and El-Rewini, 1992: 260).

CFD Concepts. The field of CFD is built on the basics of fluid dynamics. This topic is too broad to cover in a thesis background section. Instead, this section focuses on the parts of the CFD process that are important to a discussion of the parallelization of CFD applications.

The first step in the solution of a fluid dynamics problem is the generation of a representative grid for the body under consideration. A grid is a division of a continuous area into portions that are later used to represent points in a flowfield for numerical calculations. For a specific application or numerical approximation, the finer a grid (to a limiting point), the more accurate the results can be, because the accuracy of numerical schemes increases with decreased grid point spacing.

CFD grids are structured or unstructured. Structured grids are currently the most widely used. Structured grids are the easiest to conceptualize due to their regular shapes. In the simplest geometries, a structured grid consists of squares in two dimensions, or cubes in three dimensions. Once a structured grid is generated, the computer time required to solve for the flow around the body, using a specific set of equations and assumptions, is usually less than it would be using an unstructured grid. Computations using an unstructured grid require more time due to increased bookkeeping. However, unstructured grids have gained acceptance in the field of CFD. Because they consist of triangles in two dimensions, or tetrahedra in three dimensions, they require less time than a structured grid to generate for a complex body. Unstructured grids also simplify the modeling of adaptive bodies, such as a wing with an adjustable flap.

Two primary approaches used to create unstructured grids are the advancing front method and Delaunay triangulation. The advancing front method is the most mature. Delaunay triangulation, however, results in triangles that are less skewed. The advancing front method creates a grid by marching away from the body's surface and creating the grid as it progresses. The Delaunay triangulation grid is generated by repeatedly inserting points into the current grid. Any triangle whose circumcircle includes the new point is divided. Figure 1-1 shows the insertion of such a point. Delaunay triangulation creates cells in which the minimum angle of each triangle is maximized. Using Watson's algorithm, new grid points are generated within triangles whose aspect ratio is greater

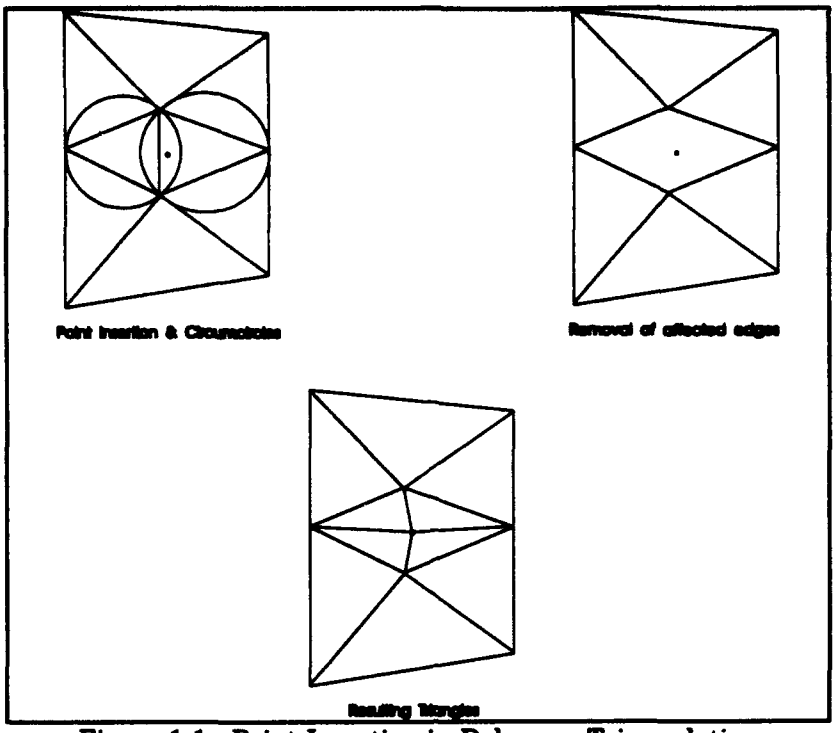


Figure 1-1. Point Insertion in Delaunay Triangulation

than a specified value (Barth, 1992:6-15). The aspect ratio of a triangle is the ratio of its circumcircle to its innercircle (see Figure 1-2). Holmes and Snyder expand on Watson's algorithm to include the area of the triangle as a criterion for point generation (Anderson, 1992:1).

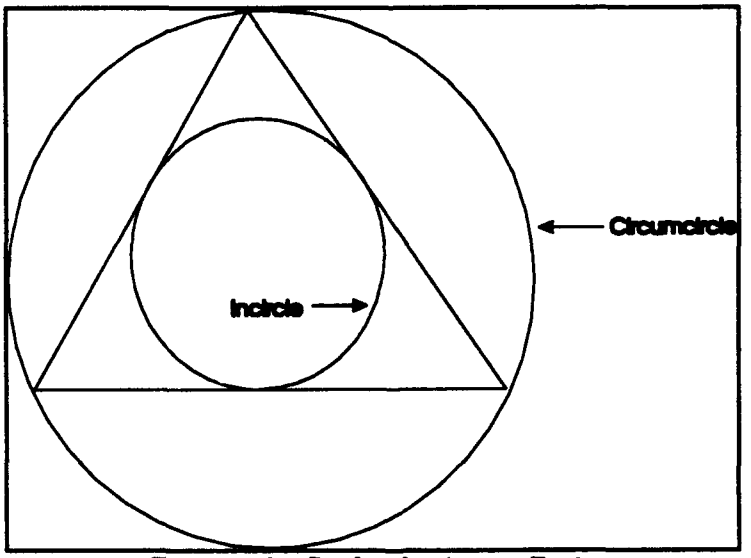


Figure 1-2. Circles for Aspect Ratio

The less skewed triangles generated by the Delaunay triangulation method lend themselves to better solution accuracy. At first glance, however, the advancing front method appears to be easier to parallelize because it involves creating an appropriate grid where there is empty space. Only the border triangles must be coordinated between processors. The border construction can even occur after the processors have completed "filling" their assigned empty spaces. Delaunay triangulation involves modifying an existing grid which involves more interprocessor communication because neighboring triangles may be located on separate processors.

CFD calculations can use either two-dimensional or three-dimensional grids. Although three-dimensional grids provide a better representation of reality, they are computationally expensive to generate. Two-dimensional grids are often used to study cross-sections of a body and in the development of numerical algorithms. After an algorithm is developed for two-dimensional grids, it can be adapted for three-dimensional grids.

The field of CFD involves the use of many different mathematical representations of flowfields. The primary sets of equations are the Navier-Stokes and Euler equations. The Navier-Stokes equations provide a method for the examination of flowfields in a broad range of circumstances. Solutions involving the full set of Navier-Stokes equations are extremely expensive in terms of computer resources and time. The time and resources required for many applications, can be limited by using subsets of these equations. The Euler equations are one such subset. The Euler equations provide an approximation of fluid flows where viscosity and heat transfer may be assumed negligible.

The driving force behind the selection of a set of equations is the specific CFD application. Applications can involve internal or external flow, laminar or turbulent flow, compressible or incompressible fluids, steady or unsteady flow, and viscous or inviscid considerations, just to mention a few. The type of CFD algorithm that is chosen is based on the equations and grid representation used for a specific application. The three most common types of CFD algorithms are finite-difference, finite-volume, and finite-element.

The recent increase in the use of finite-volume solvers is due, in part, to the evolution of unstructured grids.

Problem

This research develops a parallel algorithm to create a two-dimensional unstructured grid using Delaunay triangulation for a given simple object. The goal was to achieve linear speedup on a distributed memory MIMD parallel computer without sacrificing any accuracy provided by the sequential Watson/Bowyer algorithm currently used. The methodology used focuses on a data parallel approach because grid data can be logically distributed onto processors in uniform pieces. The operations performed on the data cannot be readily modeled with a task graph because a cell, created through Delaunay triangulation method, can have more than four neighbors. Therefore, care must be taken not to map the grid to the processors in a way that will cause the processing to be dominated by communications.

Assumptions and Scope

The field of CFD, and fluid dynamics in general, is quite broad. This research is limited to a specific type of application and the utilization of specific methods to solve the individual parts of the problem. Although this specialization limits the applicability of the algorithms, it is necessary to reduce the size of the problem and correspondingly, the time required to run the software once it is implemented. Broad algorithms can be applied to more applications, but every application solution requires a longer time than would be required using a more specific algorithm. For numerically intensive and exceedingly large problems, problem specific algorithms are often the best way to attack the problem. To cover a wide range of problems, one solution is to develop a library of specific algorithms. The grid is limited to two dimensions around a simple elliptical or cylindrical object. The flow is assumed to be external to the object. The clustering function used assumes the

areas of interest lie on or near the surface of the object. Other possible areas of interest are not considered in order to simplify the problem. To generate the unstructured grid the software uses the method of Delaunay triangulation. This section discusses the limitations of these assumptions.

An unstructured grid is a viable tool for a finite-volume solver. This type of solver involves the determination of values in every cell of the grid. It is useful for geometrically complex objects. A finite-volume solver can also provide improved results over finite-difference methods for structured grids (Beran, 1992).

The use of an unstructured grid is appropriate for a finite-volume solver. The application indicated by Smith is most likely to be applied during situations such as an aircraft landing (Smith, 1992). During a landing, the wing often changes shape due to the use of flaps. An unstructured grid is much better at adjusting to a shape change than a single structured grid. A change in shape does not require regeneration of the entire grid, only the small portion affected by the change. For ease of development the methodology uses a two-dimensional grid.

The Delaunay triangulation method assists in generating a grid whose elements are not usually highly skewed. More consistent triangles create more consistent grid elements, which are used in the numerical analysis. The shape consistency of the triangles improves the accuracy of finite-volume solvers.

An efficiency of over 70% is accepted as reasonable by the parallel computation community (see Chapter 2). The parallel computation metrics, efficiency and speedup, indicate the possible scalability of the code.

Approach

The first step in the research process was to completely understand the problem and its various parts. Part of this understanding process consisted of an extensive literature

search. The remainder came about through discussions with personnel involved with both CFD and parallel algorithms.

The main part of this research included developing, implementing, and testing algorithms for an unstructured grid generator. Several subtasks make up this larger task. The first subtask is the development of a parallel algorithm for Delaunay triangulation using software engineering techniques. The methodology included consideration of load balancing and the communication requirements of any given problem distribution. It also ensured the resulting grid was fine enough to use with the flow solver to arrive at an answer that displayed an acceptable amount of accuracy. The second subtask was the implementation of the algorithm. This subtask required consideration of which language and architecture to use for implementation of the algorithm.

The test plan consisted of several parts. First, testing involved comparing the resulting grids with others from sequential grid generators. It then involved running the results of the grid generator through a flow solver for problems with known solutions. A determination of the amount of error in the algorithm is available from these tests. An acceptable level of error is dependent on the methods reflected in the algorithms and the applications that utilize the code. Computer architectures can affect the numerical accuracy of the results as can the differences in the implementation of double-precision numbers on different types of computers. The tests also examined the differences that the addition of clustering and smoothing made to the accuracy of the results.

The metrics of speedup and efficiency can use the results of these, and similar, tests. Two additional tests were required. The first set of tests involved executing the code on a varying number of processors, while holding the problem size, or grid size, at a consistent level. The second set of tests involved holding the number of processors level and increasing the size of the problem. These additional tests provided insight into the scalability of the algorithm for the specific problem. Other tests regarding parallel

performance included static versus dynamic domain decomposition and the utilization of different domain decomposition techniques.

Required Equipment

The software developed in support of this thesis effort was hosted on the Intel iPSC/2 Hypercube. The decision to develop codes for this architecture was based on several factors. One factor was the availability of the architecture. The software development was also done on the iPSC/2 because of prior programming experience.

Summary

The process of parallelizing CFD applications included several steps. These steps were the development and implementation of algorithms for the grid generator, and testing for the accuracy of results, and the usefulness of the algorithms on parallel architectures. This thesis investigation focuses on a specific application within the field of CFD. Focusing on a specific application limits the scope of the problem to an achievable level. This research provided the first parallel grid generation algorithms for AFIT CFD research.

The next chapter provides a literature review of significant research in the area of parallel computation fluid dynamics applications. Chapter 2 lays down a foundation for the work performed in this thesis effort. Chapter 3 builds on the concepts introduced in Chapter 2. It explains the methodologies involved in grid generation, domain distribution, and software design and implementation. A discussion of both versions of the grid generation software and the recursive spectral bisection code is also included in Chapter 3. Chapter 4 presents the testing methodologies and the results achieved through the use of the methodologies included in Chapter 3. The results are presented in numerical and graphical form. Finally, Chapter 5 consists of the conclusions and recommendations resulting from the completion of this thesis research.

Chapter 2: Literature Review

Introduction

A relatively new class of computer systems, massively parallel processors, introduces the possibility of increased accuracy for CFD applications. This possibility exists because fewer assumptions need to be made to complete the calculations necessary for a given application in a limited amount of time on a parallel architecture. There is a significant research effort underway that focuses on the parallelization of CFD applications. The purpose of this chapter is to examine literature dealing with the parallelization of CFD, with an emphasis on grid generation. Much of the information in this chapter also provides a background for the methodologies described in Chapter 3.

Viability of Parallel Computers for CFD

A wide range of authors have shown the viability of using parallel computers for solving CFD applications. This section demonstrates the parallelism inherent in grid generation and discusses the results of work performed by several researchers. Many researchers compare their results to those obtained on Cray computers. The most important aspects of their work are the demonstrated speedup, efficiency, scalability, and possibilities for future improvement. Note that most of the reported research deals with flow solvers rather than grid generators because little has been done on the parallelization of grid generators because the issue of parallelizing CFD applications is relatively new.

Lohner, Cambros, and Merriam presented their successful implementation of a parallel unstructured grid generator using the advancing front method in a recent paper (Lohner, Cambros, and Merriam, 1992). Although they do not discuss their results in terms of speedup and efficiency, they discuss many important issues surrounding parallel unstructured grid generation. One important issue is the ability to parallelize the process of grid generation. The authors note that the process of grid generation is scalar because it

involves the insertion of one point after another. However, "as the introduction of a point or element only requires checking the local neighborhood for compatibility, one may introduce several (possibly many) points or elements at the same time" (Lohner, Cambros, and Merriam, 1992:33). The researchers believe "that the inherent parallelism is independent of the size of the final mesh" (Lohner, Cambros, and Merriam, 1992:33). This belief is logical because the initial grid is the same, regardless of the size of the final grid. This belief indicates that the application may not scale well.

Stagg and Carey examined a parabolized Navier-Stokes code on an nCUBE/2 with 1024 processors (Stagg and Carey, 1992). They used a series of two-dimensional grids to represent steady flow past supersonic and hypersonic vehicles. The nCUBE/2 is a MIMD machine with a hypercube configuration. They tested their algorithm using two methods. The first method kept the grid size constant and increased the number of processors (Stagg and Carey, 1992:332). These results indicated that the algorithm scales well up to 16 grid nodes per physical processor. The second test method involved increasing both the grid size and the number of processors at the same rate. This test also showed the flow solver algorithm to be scalable.

In an earlier paper, Agarwal compared the execution of two-dimensional Reynolds-averaged Navier-Stokes codes hosted on a Cray X-MP to equivalent converted codes on a 16,000 processor Connection Machine, CM-2. His "timing studies show that for smaller problems, Cray is significantly faster than CM; however, as the problem size increases, CM is likely to achieve faster results" (Agarwal, 1989:922). This demonstrates the potential of massively parallel processors.

In one of his papers, Braaten explored the implementation of a two-dimensional Navier-Stokes code on an Intel iPSC/2 Hypercube (Braaten, 1989). The application he discussed was a steady flow with both laminar and turbulent elements. Braaten executed his code on a 32-node scalar and an 8-node vector iPSC/2. During testing involving steady laminar flow "speedups relative to a single processor up to 20.2 with 32 processors are achieved,

demonstrating the parallel efficiency of the algorithm" (Braaten, 1989:949). "This is equivalent to a parallel efficiency of 63%" (Braaten, 1989:951). He compared these results to those obtained on a Cray X-MP. "With 32 Processors the performance of the iPSC/2 is about 1/6th that of a single-processor CRAY X-MP" (Braaten, 1989:951). "Results with an 8-node vector iPSC/2VX give 1/5 the performance of a single-processor CRAY X-MP, which translates to more than a tenfold improvement in cost performance" (Braaten, 1989:951). He concludes that as parallel architectures grow and become faster, his algorithm has the potential to overtake the performance available on the Cray X-MP and other sequential supercomputers. This conclusion is logical because the development of parallel architectures is still a new field, where as the development of the Cray has required many years.

In a recent paper, Hauser and Williams attempted to predict the efficiency of a Navier-Stokes code running on a 520-node Intel Touchstone Delta (Hauser and Williams, 1992). The Delta is a MIMD mesh architecture. They examined the flow about the European space plane Hermes. They developed a communication model for the Delta and compared it to the older iPSC/860. Although their discussion is purely theoretical, the results are quite promising. Based on their models, they predicted that their code can run on the Delta with an efficiency of over 95% (Hauser and Williams, 1992:57). They indicated several reasons for this phenomenon. The first is that the complex algorithm requires more computations per node. A computationally intensive algorithm is thus ideal for a MIMD machine. Secondly, the hardware speed available on the Delta allows communications to occur more rapidly than would otherwise be possible. Another reason is that each processor has 16 Mbytes of local memory. This large memory space allows a large amount of data to be stored on each node. Finally, a proper grid distribution contributes to an efficient implementation (Hauser and Williams, 1992:57).

Steven Scherr provided a discussion of experimental results of an explicit MacCormick predictor-corrector Navier-Stokes algorithm on an Intel Touchstone Delta (Scherr, 1993).

He used a three-dimensional structured grid to examine laminar flow about a delta wing. He balanced the workload across up to 507 nodes of the Delta machine. Scherr reported parallel efficiencies of 60-68% (Scherr, 1993:8-9). Note that this is not the same application or algorithm used earlier by Hauser and Williams, so the efficiencies cannot be directly compared.

During a 1989 conference, a group of researchers shared the results of their experiments regarding the use of an Euler code on a 16-node Intel iPSC/2 and a 512-node nCUBE/ten (Barszcz, Chan, Jesperson, and Tuminaro, 1989). Both systems are configured in a hypercube topology. The flow field, represented by a two-dimensional structured grid, was steady and inviscid. They included detailed results of numerous tests completed on both parallel architectures and one processor of a Cray X-MP. "The Intel iPSC/2 yields a speedup of 15.0 running from 1 to 16 processors, and the nCUBE yields a speedup of 10.1 running from 32 to 512 processors on the 256 x 128 grid" (Barszcz, Chan, Jesperson, and Tuminaro, 1989:937). The results presented by these authors demonstrate the possibilities available when using different parallel computers. A direct comparison based on 1 to 16 processors on the iPSC/2 to the nCUBE with 32 to 512 processors with the same size grid is not reliable. A more accurate comparison would compare the same number of processors on each system or correspondingly increase the size of the grid. Currently, neither of these computer configurations can compete with the Cray X-MP, on this specific application, due to their lack of maturity. As discussed earlier, however, Braaten reported the success of an 8-node iPSC/2VX configuration when compared to a Cray X-MP (Braaten, 1989). Both the computer configuration and the specific application affect the success of an experiment.

Another group of researchers explored the use of an Euler solver on an Intel Touchstone Delta (Mavriplis, Das, Saltz, and Vermeland, 1992). They used a three-dimensional unstructured grid. The original code was developed for a Cray Y-MP shared-memory system. This code included several grid strategies including single and multigrid arrangements. All the preprocessing, such as the grid generation, was completed on the

Cray. The Cray Y-MP outperformed the Delta by a factor of two for several reasons. The first is the shared memory of the Cray, which limits the required message passing. The Delta must rely on message passing for inter-processor communication. For this algorithm, the Delta achieved a communication to computation ratio of 50% which demonstrates fair efficiency for a MIMD architecture (Mavriplis, Das, Saltz, and Vermeland, 1992:137). The second reason for the lack of performance of the Delta is the lack of effective tools for distributed memory systems. As these tools become more mature, the Delta should make progress in the area of performance.

Braaten examined the use of a 32-node Intel iPSC/2VX (vector processor) running both Euler and Navier-Stokes solvers (Braaten, 1990). The application he discussed included both viscous and inviscid, transonic, compressible flows. Using the vector processors of the iPSC/2, he tested both line and point solvers. He decomposed the grid into overlapping regions, each of which was then assigned to a physical processor. Braaten tested the algorithm by performing 100 iterations on both scalar and vector processors for an iPSC/2. "With 32 processors, efficiencies of 80% are achieved with scalar processors, 71% with the line solver on vector processors, and 45% for the point solver on vector processors" (Braaten, 1990:466). These systems obtained approximately 1/4th the performance of a single processor Cray Y-MP (Braaten, 1990:467).

Researchers Long, Khan, and Sharp also examined both Euler and Navier-Stokes solvers (Long, Khan, and Sharp, 1991). Their code was based on the Three-Dimensional Euler/Navier-Stokes Aerodynamic Method (TEAM) used by Lockheed and the USAF (Long, Khan, and Sharp, 1991:660). They used both structured and unstructured three-dimensional grids on a Connection Machine (CM-2). They discussed the merits of each type of grid and the current bottleneck their generation creates. After testing the various configurations on the CM-2 and a Cray X-MP the authors arrived at three conclusions:

1) An unstructured grid code on the CM-2 is roughly as efficient as the TEAM code is on a Cray X-MP.

2) A structured grid code on the CM-2 is roughly 15 times faster than TEAM on a Cray X-MP.

3) An unstructured grid code on the CM-2 is roughly 100 times faster than an unstructured Euler code on a Cray X-MP when no gather scatter is used. (Long, Khan, and Sharp, 1991:665)

Their results demonstrate the high performance possible on a CM-2 by comparing the results to those achieved on a single-processor Cray X-MP. These results are limited to the capabilities of a SIMD machine as compared to a earlier single-processor Cray. Most parallel machines are now MIMD machines and most Cray computers have multiple processors. Although the results are limited, they do demonstrate the capabilities of massively parallel machines as applied to several different CFD solvers.

The experiences of all these researchers provide an optimistic look at the future of parallel CFD. They show that any architecture, when effectively utilized, can demonstrate promise for the future. The reported efficiencies range from 50% to 95%, but the 95% efficiency reported by Hauser and Williams was purely theoretical. Most researchers reported efficiencies around 60-70%. This thesis research attempts to match these efficiencies. The work of the researchers also demonstrates the variety of CFD applications that can make use of the computational power of parallel architectures. Further advances in this field will require more powerful parallel machines and the development of true parallel algorithms as compared to the parallelization of sequential algorithms.

Parallelizing Grid Generation

CFD grid generation is a topic of great interest to parallel and sequential computer users in the CFD field. While the development of structured grids is a relatively mature field, the development of unstructured grids is still relatively new. The development of parallel code to implement either one is even newer. Currently, most grid generation algorithms are implemented on sequential machines. This occurrence will eventually create severe bottlenecks in the CFD process (Long, Khan, and Sharp, 1991). This

statement is especially true of adaptive grids which must be regenerated when the surface changes. This section presents reasoning for choosing to use Delaunay triangulation, current research in the area of parallel grid generation, methods for inserting new points in a grid, and methods for the decomposition of the grid generation process. The research presented in this section provides a basis for the selection of methodologies used for this work.

Delaunay Triangulation. The two primary methods used to generate unstructured grids for CFD are advancing front and Delaunay triangulation. This section examines both methods and the advantages of using Delaunay triangulation as discussed by various authors.

In a paper for AGARD, Barth lists the properties of a two-dimensional Delaunay triangulation (Barth, 1992). The seven properties he discusses are:

1) Uniqueness - No four points are cocircular.

2) Circumcircle Criteria - The circumcircle of every triangle is point free.

3) Edge Circle Property - There exists some circle passing through the endpoints of each and every edge which is point-free.

4) Equiangularity Property - The minimum angle of the triangulation is maximized.

5) Minimum Containment Circle - The maximum containment circle over the entire triangulation is minimized.

6) Nearest Neighbor Property - An edge formed by joining a vertex to its nearest neighbor is an edge of the Delaunay triangulation.

7) Minimal Roughness - A Delaunay triangulation is a minimal roughness triangulation for arbitrary sets of scattered data. (Barth, 1992:6-12 to 6-14)

These properties make a Delaunay triangulation ideal for CFD flow field calculations because they increase the regularity of the grid.

In his book, *Automatic Mesh Generation*, George describes both advancing front and Delaunay triangulation methods for the generation of numeric meshes. The algorithms he presents are sequential, but the concepts behind the methods are the same. He states that

"nowadays, [Delaunay triangulation] seems to have the most general application" (George, 1991:32).

In a recent NASA Technical Memorandum, Anderson described his use of Delaunay triangulation for the generation of an two-dimensional unstructured grid (Anderson, 1992). He chose to use Delaunay triangulation because "the resulting meshes are optimal for the given point distribution because they do not usually contain many extremely skewed cells" (Anderson, 1992:1).

Parallel Grid Generators. The parallel unstructured grid generators developed by Arthur and Lohner, Camberos, and Merriam both use the advancing front method (Arthur and Bockelie, 1993) (Lohner, Cambros, and Merriam, 1992). There is not any research currently available that deals with the Delaunay triangulation method of parallel unstructured grid generators.

A parallel implementation of an unstructured surface grid generation program, VGRIDSG, was recently developed for NASA by Computer Sciences Corporation (Arthur and Bockelie, 1993). This program is hosted on a cluster of Silicon Graphics IRIS 4D workstations. It uses the advancing front method of unstructured grid generation. In their report they discussed the speedup achieved by the subroutine *frontuv*, which is the parallelized portion of the original VGRIDSG program, and the program as a whole. The tests were run on 1, 2, 3, 4, 6, and 12 processors. For the generation of a surface grid of a simple cube "the complete program and subroutine *frontuv* approach maximum speed ups of 8.1 and 6.2, respectively" (Arthur and Bockelie, 1993:8) For the generation of a surface grid for a Mach 3.0 High Speed Civil Transport "the speed up of subroutine *frontuv* is asymptotic to about 3.4 and the overall speed up is asymptotic to approximately 3.0" (Arthur and Bockelie, 1993:9). Their research demonstrates the possibilities of parallel grid generators, but it is limited to experiences on a group of five different IRIS computers connected via an Ethernet. Any system which makes use of individual computers

connected via a network, such as Ethernet, involves higher communication costs than processors contained within a single system unit.

Lohner, Camberos, and Merriam implemented their parallel unstructured grid generator on an Intel Hypercube (the specific machine is not mentioned) (Lohner, Cambros, and Merriam, 1992). Like Arthur and Bockelie, they developed an advancing-front grid generator. They also included a clustering function and a smoother. "Three numerical examples were presented to demonstrate the parallel grid generation and parallel smoothing processes for subdomains of arbitrary shape and complex geometries in two dimensions" (Lohner, Cambros, and Merriam, 1992: 44). The paper does not include timing results of the software. Unfortunately, this does not provide any information on the capabilities or potential of parallelizing the grid generation process.

Although there is currently a great deal of work being done on grid generators using Delaunay triangulation, the research is not in the area of parallel computing.

Point Insertion. The method used to add new points in a Delaunay triangulation CFD grid can affect the ability to parallelize an algorithm. The insertion of new points into a grid consists of two steps. The first step is to determine where the new point needs to be located. The second step is to determine how to integrate the new point into the existing triangulation. This section examines both parts of point insertion and extensions to the basic methods as discussed by various authors.

Barth discussed a number of point insertion methods that result in a two-dimensional Delaunay triangulation (Barth, 1992). He defined four groups of algorithms: "Incremental Insertion Algorithms, Divide and Conquer Algorithm, Tanemura/Merriam Algorithm, and Global Edge Swapping (Lawson)" (Barth, 1992:6-14). The Incremental Insertion algorithms included three separate algorithms: Bowyer's, Watson's, and Green and Sibson's (Barth, 1992:6-14). Bowyer's algorithm is based on the insertion of points into a Voroni diagram. A Voroni diagram is the dual of a Delaunay triangulation. After a point is inserted, any edges within the territory of the new point are deleted and reconnected to include the new

point. Watson's algorithm is similar, but is based on the insertion of points into a Delaunay triangulation. Once a point is inserted, the edges nearest the point of all triangles whose circumcircle includes the new point are deleted. The points making up these edges are then connected to the new point to form new edges. Green and Sibson's algorithm is similar to Watson's, except that the retriangulation is accomplished via edge swapping. The Divide and Conquer Algorithm discussed by Barth assumes that the locations of all new points are specified a priori (Barth, 1992:6-16). The resulting "cloud of points" is then triangulated to create a Delaunay triangulation (Barth, 1992:6-16). The Tanemura/Merriam algorithm uses an advancing front concept to develop a Delaunay triangulation (Barth, 1992:6-17). The Global Edge Swapping algorithm, developed by Lawson, "assumes that a triangulation exists (not Delaunay) then makes it Delaunay through application of edge swapping such that the equiangularity of the triangulation increases" (Barth, 1992:6-18).

In his paper, Anderson mentioned four methods for determining the location of a new point to be added to a grid: "a priori by generating points about individual components with structured grids, subdividing existing quadrilateral cells using a quadtree encoding method, embedding the geometry into a Cartesian grid," and an approach by Holmes and Snyder (Anderson, 1992:1). He used the approach developed by Holmes and Snyder based on the aspect ratio and cell area of the triangles making up the grid. This approach creates a grid which is generally "too coarse to be used for obtaining accurate flow-field solutions" (Anderson, 1992:1). To correct this situation, Anderson extended Holmes and Snyder's algorithm to include a function to cluster cells in regions of interest (Anderson, 1992:1).

Although a grid generated using Delaunay triangulation is considered optimal, "in certain regions of the mesh abrupt variations in element shape or size may be present" (Lohner, Camberos, and Merriam, 1992:40). These variations can be overcome through the

use of a smoothing function. Lohner, Camberos, and Merriam implemented a parallel Laplacian smoother for an unstructured grid in the following manner:

1) Subdivide the mesh into subdomains

2) For each timestep or smoothing pass:
   - Smooth each subdomain separately;
   - Exchange boundary information;

3) Assemble the result. (Lohner, Camberos, and Merriam, 1992:40)

This is efficient as long as the subdomains are the same size, otherwise the load balancing causes some processors to become idle while waiting to exchange boundary information with processors which have a larger computational load.

Domain Decomposition. The decomposition of the grid domain can impact the speedup and efficiency obtained by an algorithm. Grid decomposition of CFD grids has been approached in several ways. This section examines the general decomposition methods presented by several authors researching CFD algorithms and then discusses more detailed methods for the decomposition of unstructured CFD grids.

Ramesh Agarwal experimented with the adaptation of a two-dimensional Reynolds-averaged Navier-Stokes code for a 16,000 processor Connection Machine, CM-2 (Agarwal, 1989). Agarwal assigned each element of the grid to a unique processor when the number of elements was less than or equal to the number of physical processors. When the number of elements is greater then the number of processors, "each processor acts as a virtual processor so long as the memory size is not exceeded" (Agarwal, 1989:921). This mapping provided the best load balancing possible.

Hauser and Williams' grid partitioning provided a balanced approach by dividing the structured grid into equal size blocks. However, "to simulate the irregularity caused by the geometry of the solution domain, we have made random assignments of the blocks to the processors of the machine" (Hauser & Williams, 1992:56). Each processor completed approximately the same amount of work. This balance limited the amount of time any

processor was idle while waiting for communications from the other processors. The communication performance could have been reduced, however, by using a mapping of blocks to processors that limited the distance between adjacent blocks.

More detailed domain decomposition strategies for unstructured grids are discussed by Barth in his paper for an AGARD-sponsored course (Barth, 1992). In his paper, Barth discussed Coordinate Bisection, Cuthill-McKee, and Spectral Partitioning algorithms for domain decomposition and their success as used in parallel CFD calculations. For detailed explanations of these algorithms refer to Barth's paper. This section discusses only the results of their use. Coordinate bisection "is very efficient to create but gives sub-optimal performance on parallel computations owing to the long message lengths that can routinely occur" (Barth, 1992:6-9). The results of using the Cuthill-McKee algorithm "indicate a performance on parallel computations which is slightly worse than the coordinate bisection technique" (Barth, 1992:6-9 to 6-10). According to Barth, "the objective of the spectral partitioning is to divide the mesh into two partitions of equal size such that the number of edges cut by the partition boundary is approximately minimized" (Barth, 1992:6-10). "We found that parallel computations performed slightly better on the spectral partitioning than on the coordinate bisection or Cuthill-McKee. The cost of the spectral partitioning is very high" (Barth, 1992:6-10).

It is possible to reduce the cost of spectral partitioning by using parallel computers to run the algorithm. Leete, Peyton, and Sincovec developed a parallel recursive spectral bisection mapping tool on an Intel iPSC/860 (Leete, Peyton, and Sincovec, 1993). They expanded the basic RSB method to include the ability to distribute the work evenly, even when the amount of work required for each element is uneven, and the ability to distribute the elements across p processors, where p is not a power of two (see Chapter 3 for further details). Their "parallel implementation's capacity to make good use of the extra processors is quite modest on the Intel iPSC/860" (Leete, Peyton, and Sincovec, 1993: 926). The authors do not provide speedup information. They do acknowledge that their research is

only the first step towards an efficient implementation of a parallel RSB tool (Leete, Peyton, and Sincovec, 1993: 927). Because RSB has demonstrated superior partitioning, work on parallel RSB tools will undoubtedly be continued by these researchers and others interested in achieving an optimal partitioning.

Lohner, Camberos, and Merriam listed the following domain splitting algorithms in their paper: "Simple Cartesian splitting, Quadtree/Octree splitting, and Background grid splitting" (Lohner, Cambros, and Merriam, 1992:34). Of these options, they chose to implement Background grid splitting for three reasons:

> - Background grid generation is extremely cheap. Thus, a fairly fine initial background grid can be assumed. This, in turn, allows division of the background grid into subdomains of nearly equal size.
>
> - The most demanding task for grid generation is the adaptive regeneration. Thus, again a fairly fine initial grid can be assumed.
>
> - Three major pieces of software, the grid smoother, the field solver, and the grid generator can all use the same algorithm to generate subdomains. This reduces software development costs. (Lohner, Cambros, and Merriam, 1992:35)

These reasons are very applicable to the unstructured grid generator developed as part of this research.

Environmental Issues

The environment available on a specific hardware platform can greatly affect the development, and even success, of a parallel CFD algorithm and its implementation. The environment consists of many items, including the architecture and programming language.

Probably the most important environmental issue involved in the parallelization of any application is the architecture of the computer system. Which architecture is the "best" is still not agreed on. The "best" architecture often proves to be that which best matches the application. The earlier discussion focusing on the results of work done on parallel CFD indicated the performance of algorithms on specific architectures.

Another environmental factor is the programming language used for an application. Although this factor is somewhat limited by the languages implemented on the hardware, most common languages such as Fortran, C, Ada, and Lisp are available on most hardware platforms. Once again, the "best" language often depends on the application. When given an option, most scientific programmers use Fortran or C. These languages are well suited for numerical algorithms. This fact is especially true of Fortran, which was developed for scientific programmers. There is a push in the software engineering field to use object-oriented programming (OOP) methods. In a paper presented at the *1989 Conference on Hypercubes, Concurrent Computers, and Applications*, Angus and Thompkins compared the use of OOP methods, using C++, to Fortran or C for the development of CFD code (Angus and Thompkins, 1989). The application they used is based on the Euler equations. Following a discussion of the individual implementations, they concluded that both methods are effective. They also concluded that the use of OOP techniques "provides a major enhancement of programmer productivity and portability between all machine architectures" (Angus and Thompkins, 1989:929). However, the research also demonstrated that "improvements in execution efficiency must be achieved before these languages' methods can supplant FORTRAN and C for large scientific computations" (Angus and Thompkins, 1989:929). Many other experts agree that much work remains to be done in the area of OOP before it can compete with the efficiency of languages such as Fortran and C.

The dominance of Fortran and C as the languages of choice for scientific programmers is continually fueled by new developments in parallel versions of these languages. In a recent paper, Olander and Schnabel examined the use of parallel languages such as DINO, a parallel version of C, and Fortran D (Olander and Schnabel, 1992). They concluded that parallel languages are an important addition to the toolbox of scientific parallel programmers, but there is still work to be done to make it easier to obtain an optimal parallel form of CFD algorithms (Olander and Schnabel, 1992:283). Many vendors

continue to develop new languages and new versions of popular languages to aid parallel programmers. High Performance Fortran (HPF) is currently one of the hot topics in the parallel computing field.

As the field of parallel computing matures, researchers are exerting a significant effort to develop tools to aid in porting and developing parallel code. There is a great need for mature parallel software tools. A recent article by Mavriplis, Das, Saltz, and Vermeland discusses these needs (Mavriplis et al., 1992). After experiencing difficulties developing CFD codes based on the Euler equations for a variety of parallel architectures, these researchers concluded that "as massively parallel software tools become more mature, the task of developing or porting software to such machines should diminish" (Mavriplis, 1992). Until adequate software tools are available, the process of parallelizing CFD code will grow slowly.

The parallel environment plays an important role in the development of effective, efficient codes. The architecture issues are being addressed by researchers in cooperation with the High Performance Computing and Communications Program which was developed by the National Sciences Foundation (Committee on Physical, Mathematical, and Engineering Sciences, 1991). Researchers are also aggressively addressing the software issues. Many commercial companies are quickly coming to the realization that a good architecture alone will not survive in this competitive field.

Summary

The research of the authors presented in this chapter provides background information and reasoning for decisions made in the methodology used in this thesis effort. The results achieved by numerous researchers demonstrate that the goal of parallelizing the CFD grid generation process is viable. The research also provides the background for choosing Delaunay triangulation as a method for generating unstructured CFD grids. The use of Watson's point insertion algorithm and Holmes' and Snyder's point generation algorithm

are supported. The use of a RSB methodology for the distribution of the grid is also justified. Finally, the research also provides a basis for choosing a language and architecture for the implementation of the grid generator.

Chapter 3: Methodology

Introduction

This chapter explains the methodologies the software used for each portion of the parallel unstructured grid generator. The grid generator for the unstructured CFD grid is based on Delaunay triangulation as discussed in earlier chapters. A comprehensive discussion of the methodologies involved include methodologies for each step in the grid generation process as well as methodologies for domain decomposition and parallelization. Also included in this chapter is a discussion of the parallel design and development process used to define and create the software.

Grid Generation

Several steps comprise the grid generation process. Figure 3-1 shows the decision tree used to select methodologies for the various parts of the grid generator. The methodologies chosen by the author, are indicated by the continuation of the tree from that point. These choices are based on the comprehensive literature search presented in Chapter 2. The unstructured grid generator uses the Delaunay triangulation method of grid development and the incremental insertion algorithm from Watson. The positions of new points are obtained using a methodology developed by Holmes and Snyder, which bases point generation on aspect ratio and cell area of the individual triangles already present in the grid. A discretized surface definition is used to maintain surface data integrity. Finally, clustering and smoothing functions are added to the basic generator. Following a description of the surface generator, this section describes each portion of this methodology description.

Surface Generation. The first step in defining a grid is the generation of the surface and far-field boundaries for the object of interest. This grid generator uses simple object shapes: ellipses and cylinders. These shapes enable the software to generate grids for

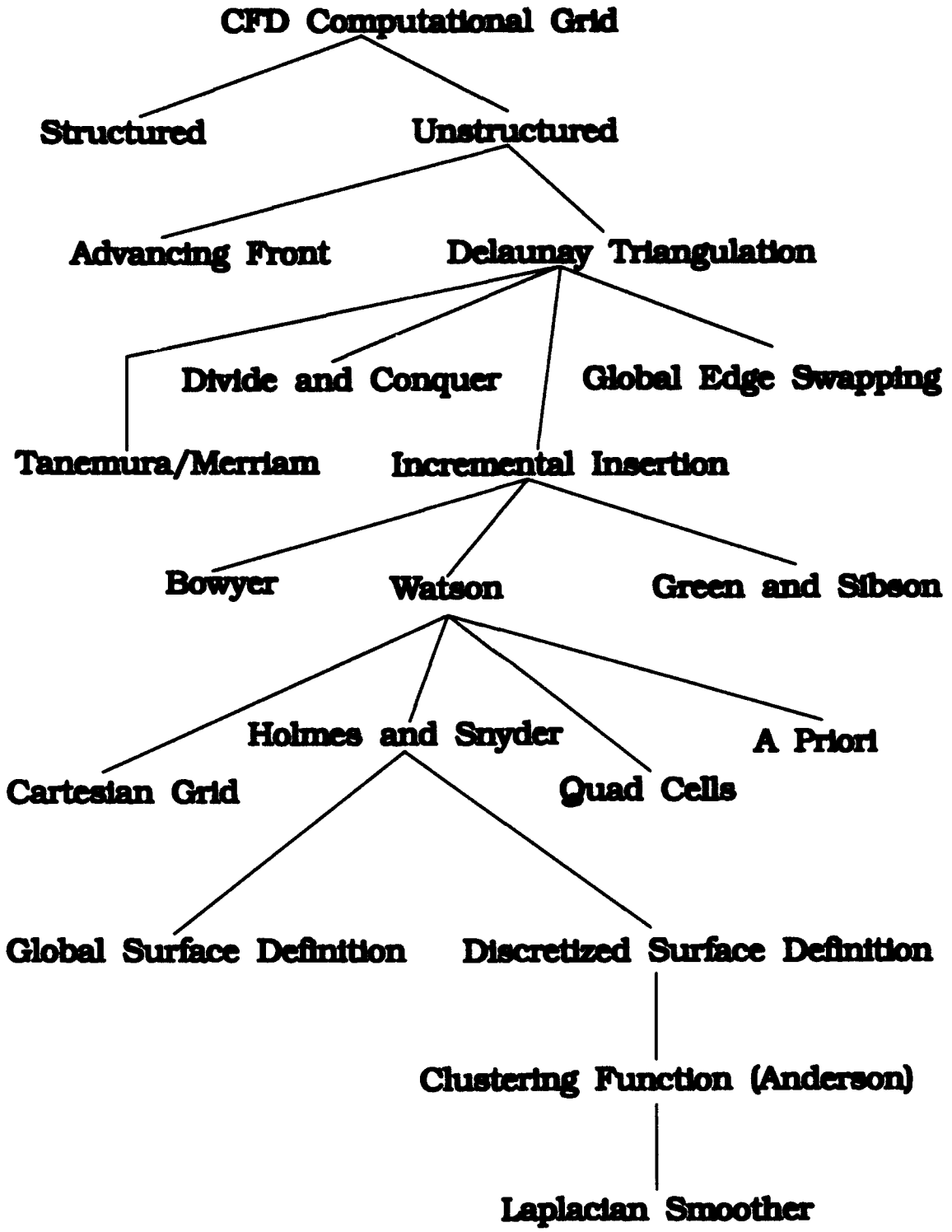


Figure 3-1. Methodology Decision Tree

several objects without introducing the problems of complex shapes. The process of creating the surface shapes is based on the number of desired points on the surface and a mathematical model of the shape desired. The far-field boundary is defined as a circle about the object with a radius defined by the user to ensure the boundary is far enough away from the object to be considered to be in the free stream. The code used to generate the shapes mathematically is a variation on Smith's work (Smith, 1992).

Initial Grid Generation. To use Watson's point insertion algorithm, the design must first define a basic grid. The initial grid consists of two triangles generated by connecting two corner points of a square. The length of the sides of the square are twice the radius of the far-field boundary plus two. These lengths create a square grid one unit outside the far field boundary that does not interfere with the generation of the initial surface grid.

Point Insertion. The surface and far-field boundary points are inserted sequentially. The points are read in from files created by the surface generation portion of the code. Due to the small initial grid, and because there are relatively few boundary points, a parallel implementation of the insertion of these points is inappropriate. An evenly balanced load would assign few triangles per processor and communication requirements would dominate the overall time required for the insertion of the boundary points.

Following the insertion of the boundary points, new points are generated based on the aspect ratio of each triangle in the grid. If the aspect ratio (see page 1-6) of a triangle is greater than 1.5, a new point is inserted in the grid at the location of the circumcenter of the triangle. This methodology is based on the approach developed by Holmes and Snyder (see page 2-10) (Anderson, 1992). Holmes and Snyder's approach also includes point insertion based on the area of an individual triangle. This addition provides a method for reducing large triangles that may meet the aspect ratio requirements, but are exceedingly large for determining flow field values.

Boundary Definition. The insertion of any point that would break up any cell making up the interior of the object, or the area exterior to the far-field boundary is rejected. This

methodology is based on a discretized view of the surface definition. It provides a method for maintaining the integrity of the surface and far-field boundary data. This methodology is appropriate for well-defined surface and far-field boundaries. Any point insertion that could affect the definition of a boundary could have a negative affect on the resulting calculations for a well-specified boundary.

Clustering. Holmes and Snyder's methodology of point generation successfully generates a grid consisting of triangles that are not highly skewed; however, the resulting grid is too coarse to be used for accurate flow field calculations. One method for overcoming this limitation is to implement a clustering function in addition to the point generation methodology of Holmes and Snyder. This clustering function acts to increase the number of cells in regions of interest such as the leading and trailing edges of an airfoil. Clustering can be computationally expensive, so there is a trade-off between increasing the positive effect of the grid and the time required to create that effect. Anderson discussed a function in his NASA memorandum that clusters cells near the surface of the object (Anderson, 1992:5). His function, ϕ, is based on the area of the cell and a weighting function, which is based on the cell's distance away from the object surface:

$$\phi(A, d) = A * f(d) \tag{3-1}$$

where

d = *distance from the cell center to the nearest point on the surface*

A = *cell area*

This variable is used to divide cells who have a value of ϕ(A,d) greater than the average plus the standard deviation of the initial distribution. The function is evaluated for each cell to provide a point insertion criterion.

Anderson uses the following weighting function:

$$f(d) = \frac{1}{1+e^{\beta(d-d_0)}} \tag{3-2}$$

where

d_0 = *distance from the surface where clustering will begin to occur*

β = *indication of how fast clustering will occur*

According to Anderson, "three or four repetitions in which β is gradually increased leads to grids with good clustering near the surface" (Anderson, 1992:6). The points generated through the clustering function are inserted into the grid using Watson's methodology as described above.

Smoothing. One final step is required to produce a grid that improves the accuracy of solutions computed with the standard, finite-volume algorithms. This step is the inclusion of a smoothing function. A smoothing function eliminates abrupt variations in size and shape that may be present in neighboring cells. The most commonly used smoother is a Laplacian smoother. The software design implements this smoother because it is prevalent in current literature. This function repositions points according to the following equations (Anderson, 1992:6):

$$x_i^{n+1} = x_i^n + \frac{\omega}{n}\sum_{k=1}^{n}(x_k - x_i)$$
$$y_i^{n+1} = y_i^n + \frac{\omega}{n}\sum_{k=1}^{n}(y_k - y_i) \tag{3-3}$$

where

x_i, y_i = *x and y coordinates for point i*

ω = *relaxation factor*

n = *number of iterations*

Domain Distribution

The grid generation software developed for this thesis is hosted on a MIMD architecture. The Intel iPSC/2 has a hypercube topology. A MIMD architecture is utilized because of the coarse grain of the problem domain. The insertion of each point requires a unique, large set of calculations. Although the specific calculations, such as the calculation of the aspect ratio of a triangle, are the same, the number of times a given equation must be calculated varies from point to point. If this algorithm were to be efficiently implemented on a SIMD machine, the calculations for each point insertion would have to be the same. However, this constraint wastes valuable time in the calculation of variables of neighbors for a triangle that has few affected neighbors. Another reason to use a MIMD architecture is because of the limited number of inter-processor communications required. This reason is especially valid for the basic methodology of domain decomposition discussed next.

The grid generation software uses two separate domain decomposition methodologies for distributing the grid cells across the processors of the parallel system. The first is a simple methodology using the far-field boundary, which is a circle, as a basis. The 360-degree circle is divided into p sections, where p is the number of processors being used. Any cell that consists of two or more points located in a given section is assigned to that section. If one of the points lies directly on a boundary, the third point is considered. If one point lies on a boundary and the remaining points are in different sections, then the software determines the circumcenter of the triangle and assigns the triangle to a section based on the location of the circumcenter. Another possible method for determining a cell's section is to rely solely on the location of the circumcenter of the cell and assign the cell to the section to which the circumcenter is located. This method would require additional calculations that are not required by the first method. This requirement is not as important when using static distribution, where the calculations are done only once. It

could be a major factor, however, if dynamic load balancing is used and the calculations are done repeatedly.

Using this simple decomposition method allows the location of the section boundaries to vary. Therefore, new points can be added in a section within cells that do not affect cells assigned to other processors. When that process is complete, the section boundaries can be moved to group cells that were near section boundaries. This allows the boundary cells to be divided based on the point generation criteria, thereby eliminating much of the inter-processor communication that would otherwise be required during point insertion. It does, however, increase the number of communications required between iterations, because half of a processor's cells are sent to its neighbor after an iteration is complete. This methodology is not completely scalable, because as the number of sections, or processors, increase, the probability that a cell lies on a boundary increases. Eventually, the sections are so small that a cell can be continuously located on a boundary even after the shuffle is complete. In this case, the cell never has a chance to be modified. Another communication structure which provides for interprocessor communication during point insertion allows this decomposition to scale better. It is still not completely scalable, however, since the load for processors whose sections include the leading and trailing edges of an elliptical object would be greater than the others.

Currently, there is little need to parallelize the initial grid distribution portion of the grid generation software using this distribution method. The full calculations are done only once at the beginning of the program, at which time there are relatively few cells. If the calculations were required for a large number of cells, parallelization would be required to achieve the highest speeds. The intermediate shuffle of domain information is completed in parallel because each processor has control of its own cells and has the ability to determine which cells to keep and which to pass to its neighbor.

The second domain decomposition methodology employed by the grid generator is spectral partitioning, also called recursive spectral bisection. This methodology provides a

balanced distribution of cells by recursively dividing the number of cells in half. It also generates a distribution that requires a minimum number of messages to be passed between processors by minimizing the number of cells cut. The spectral partitioning methodology is implemented in the grid generation software as an addition to replace the basic methodology described above. The sequential algorithm for spectral partitioning, as presented by Barth, consists of the following steps:

1) Calculate the matrix L associated with the Laplacian of the graph.

2) Calculate the eigenvalues and eigenvectors of L.

3) Order the eigenvalues by magnitude, $\lambda_1 \leq \lambda_2 \leq \lambda_3 \ldots \lambda_N$.

4) Determine the smallest nonzero eigenvalue, λ_f and its associated eigenvector x_f (the Fiedler vector).

5) Sort elements of the Fiedler vector.

6) Choose a divisor at the median of the sorted list and 2-color vertices of the graph which correspond to elements of the Fielder vector less than or greater than the median value. (Barth, 1992:6-10)

The Laplacian matrix is defined as $L = D - A$, where D is the diagonal matrix with entries equal to the degree of each vertex, $D_i = d(v_i)$, and A is the adjacency matrix of the mesh. The Laplacian matrix is also defined as follows (Hendrickson and Leland, 1993: 954-955):

$$L_{i,j} = \begin{cases} -1 \;\; if \;\; (V_i, V_j) \;\; \epsilon E \\ d_i \;\; if \;\; i=j \\ 0 \;\; otherwise \end{cases} \tag{3-4}$$

where

$L_{i,j}$ = *value at row i, column j*

V = *vertex*

E = *set of edges*

d_i = *diagonal matrix element*

This sequence of steps is repeated n/2 times, where n is the number of partitions desired. This algorithm is applicable only when the number of nodes is a power of two. Several researchers from Oak Ridge National Laboratory developed parallel code which is also applicable for situations where the number of nodes is not a power of two. The variation is accomplished as follows:

1. If $|P|$ is even, then partition P into P_1 and P_2, where $|P_1| = |P_2|$; if $|P|$ is odd, then partition P into P_1 and P_2, where $|P_1|$ and $|P_2|$ differ by one.

2. Determine the Fiedler "cut" point for which

$$\left| \frac{\sum_{v_k \in V_1} w_k}{|P_1|} - \frac{\sum_{v_k \in V_2} w_k}{|P_2|} \right| \qquad (3\text{-}5)$$

is as small as possible for the resulting partition $V_1 \cup V_2 = V$; then partition V into V_1 and V_2. (Leete, Peyton, and Sincovec, 1993)

The original definition of recursive spectral bisection (RSB) also limits the mapping by assuming that every vertex is weighted equally. This is not the case when vertices require different amounts of processing. The Oak Ridge researchers also addressed this problem. They assign a weight, w_k, to each vertex and choose the middle element of the Fiedler vector "with respect to the weights, so that

$$\left| \sum_{v_k \in V_1} w_k - \sum_{v_k \in V_2} w_k \right| \qquad (3\text{-}6)$$

is as small as possible" (Leete, Peyton, and Sincovec, 1993: 925).

The spectral partitioning methodology can be used for either static or dynamic domain decomposition. A static implementation provides a satisfactory initial grid decomposition. If the grid develops in a manner that maintains an even load balance, a static decomposition is ideal, because it limits the number of interruptions to the generation process. This balanced development is not likely for a spectrally partitioned grid, however. The partitions are likely to group far-field and surface cells separately. The partitions consisting of cells located near the surface gain additional cells more rapidly than the

partitions consisting of far-field cells. This disparity occurs due to the introduction of a clustering function as described previously. Such a disparity eliminates a proper load balance and result in the inefficient use of some processors. This disparity can be overcome through the use of dynamic domain decomposition. Dynamic decomposition requires the spectral partitioning methodology to be repeated during the grid generation process. The repetition can occur as frequently as after every iteration of the point insertion algorithm. This frequent repetition can result in high communications overhead and long periods of grid generation interruption. An optimal solution lies between this extreme and a static distribution. Due to the time required by RSB, several iterations should occur before the domain is redecomposed.

Software Design

The initial software design included the basic functioning required for grid generation. The first step in the design process was to develop an object model of an unstructured grid for CFD applications. The basic unstructured grid is composed of points, and triangles made up of these points. These relationships are depicted in the object model in Figure 3-2. This initial model aids in the definition of the functionality of the resulting software. The model provides a basis for determining the data structures and functions required for the software.

The decision to use Fortran as the programming language for the software was based primarily on the numerical capabilities of Fortran. Most sequential software in the area of CFD is also written in Fortran. The use of Fortran as a programming language limits the object-oriented programming possibilities. The software is therefore designed functionally. The design is a top-down design that breaks down each portion of the functional characteristics of the system. This functional decomposition is transform-centered (Gane and Sarson, 1979: 187-189). All the subroutines report modified information on the

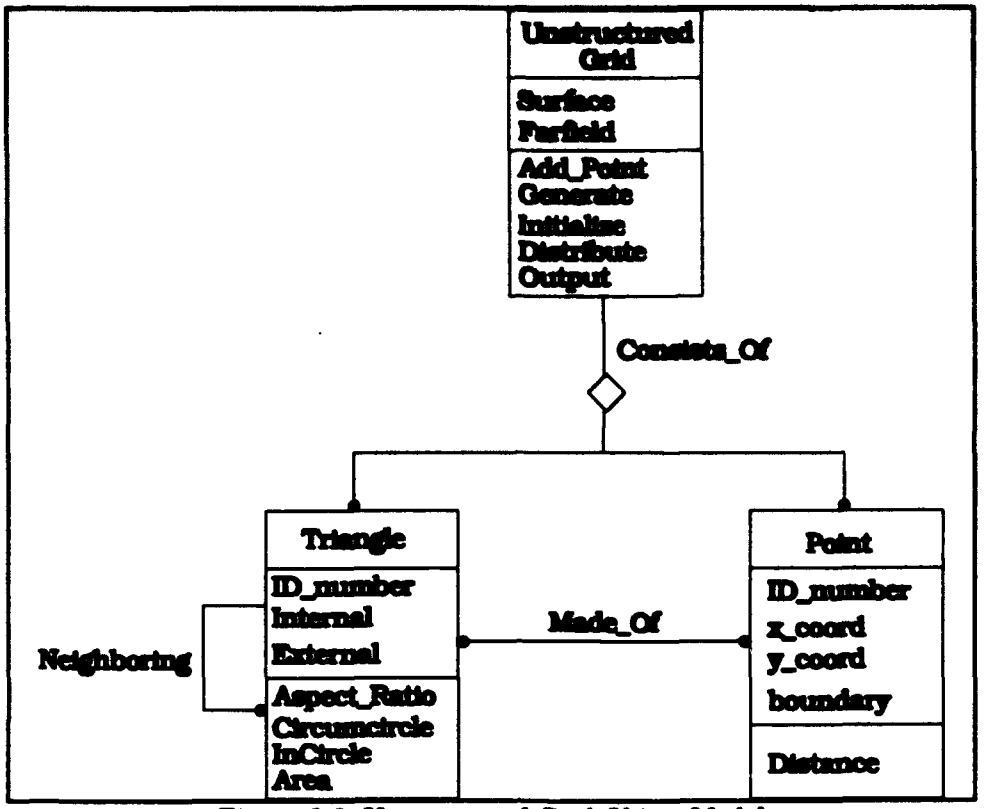


Figure 3-2. Unstructured Grid Object Model

triangles and points to the main program which then uses the information to drive the next subroutine.

The software design includes elements that are considered only because the software implementation is parallelized. The first consideration is the problem decomposition. The problem is decomposed along data boundaries. Each element of the grid, in this case each triangle, is an important data item. Operations are performed on each triangle individually. Control decomposition methods are not appropriate in this case because the same, limited number of operations are performed on each triangle. This functionality lends itself to data decomposition much better than control decomposition. To ensure proper load balancing, the distribution methodology was carefully considered. As described previously in this chapter, a simple 360 degree methodology was used to achieve an adequate load balance.

The relationships between the host and nodes comprising the iPSC/2 system were examined to determine the functionality assigned to each. Although the nodes performed the operations on the specific triangles during the parallel generation and clustering portions of the algorithm, there was a need to determine the number to be assigned to a new triangle or point. This function was assigned to the host. The function of determining and solving deadlock situations was also assigned to the host. This assignment allowed each node to report possible deadlock situations to a central point, the host. Finally, the host was assigned the responsibility of determining when all nodes were done modifying their assigned triangles. Once again, this assignment gave the nodes a central point of control. During the smoothing portion of the algorithm the nodes calculate the values of variables for the points assigned to it. The host gathers the results from the nodes and determines the values of global variables such as averages and standard deviations.

The functional design of the software is shown in Figure 3-3. An asterisk in the upper left hand portion of a box indicates that the module is broken out further in another portion of the diagram. The system subroutines used by the software are not included in the design diagrams. For information regarding the system routines for the Intel iPSC/2 refer to the *iPSC/2 Fortran Programmer's Manual* (Intel, 1988).

Version Descriptions. The implementation of the CFD unstructured grid generator included two different versions. In Version 1, the basic 360-degree grid distribution method was implemented without interprocessor communication during point insertion. The software design of the first version is depicted in Figure 3-3, with expanded versions of the InitGrid and Smoother designs shown in Figure 3-4. Version 2 used the same distribution method, but included interprocessor communication during point insertion. Finally, a RSB grid mapping tool for domain decomposition from Oak Ridge National Laboratory is included in the discussion (Leete, Peyton, and Sincovec, 1993). The design of the RSB code is shown in Figure 3-5.

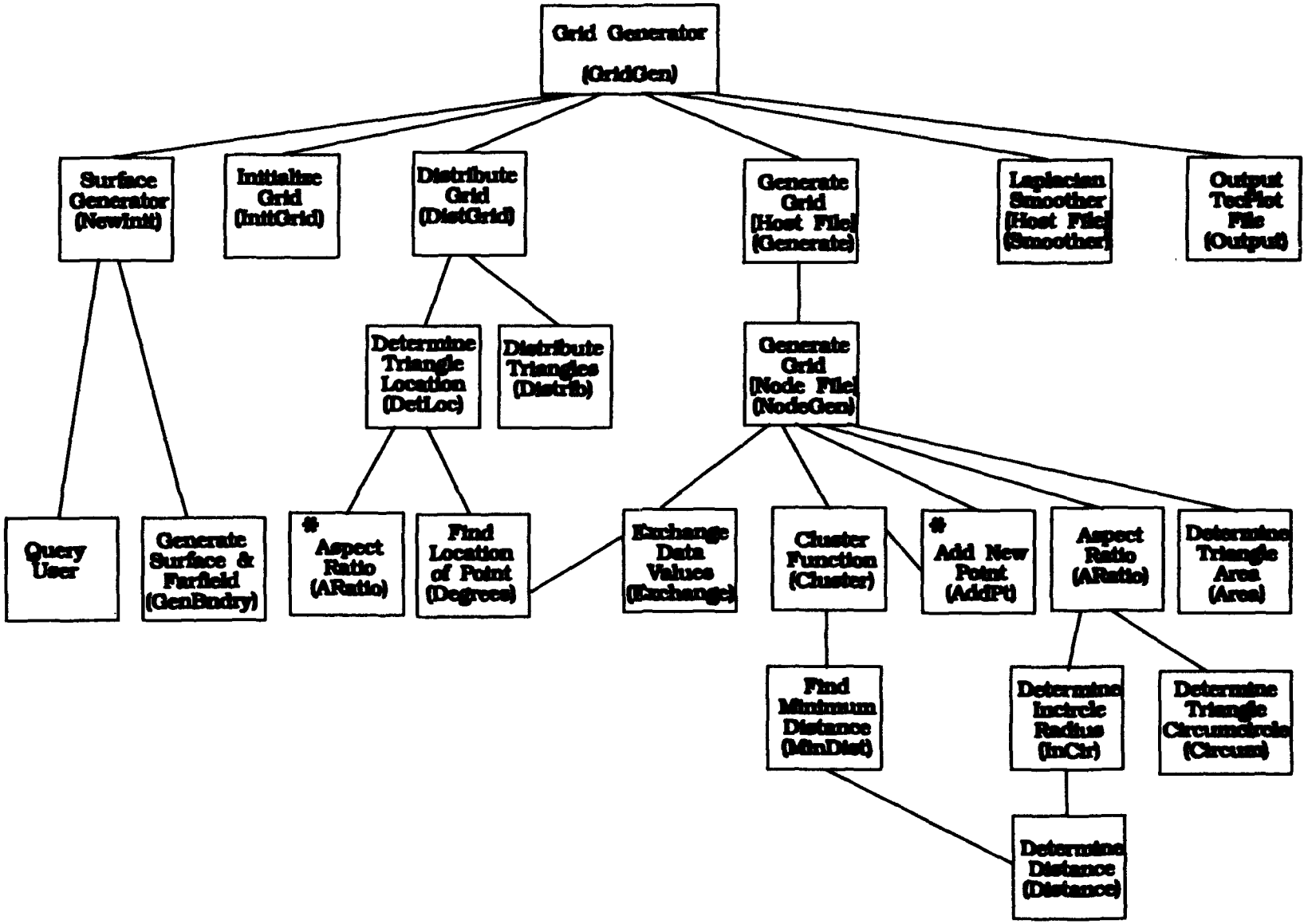


Figure 3-3. Grid Generator Software Structure

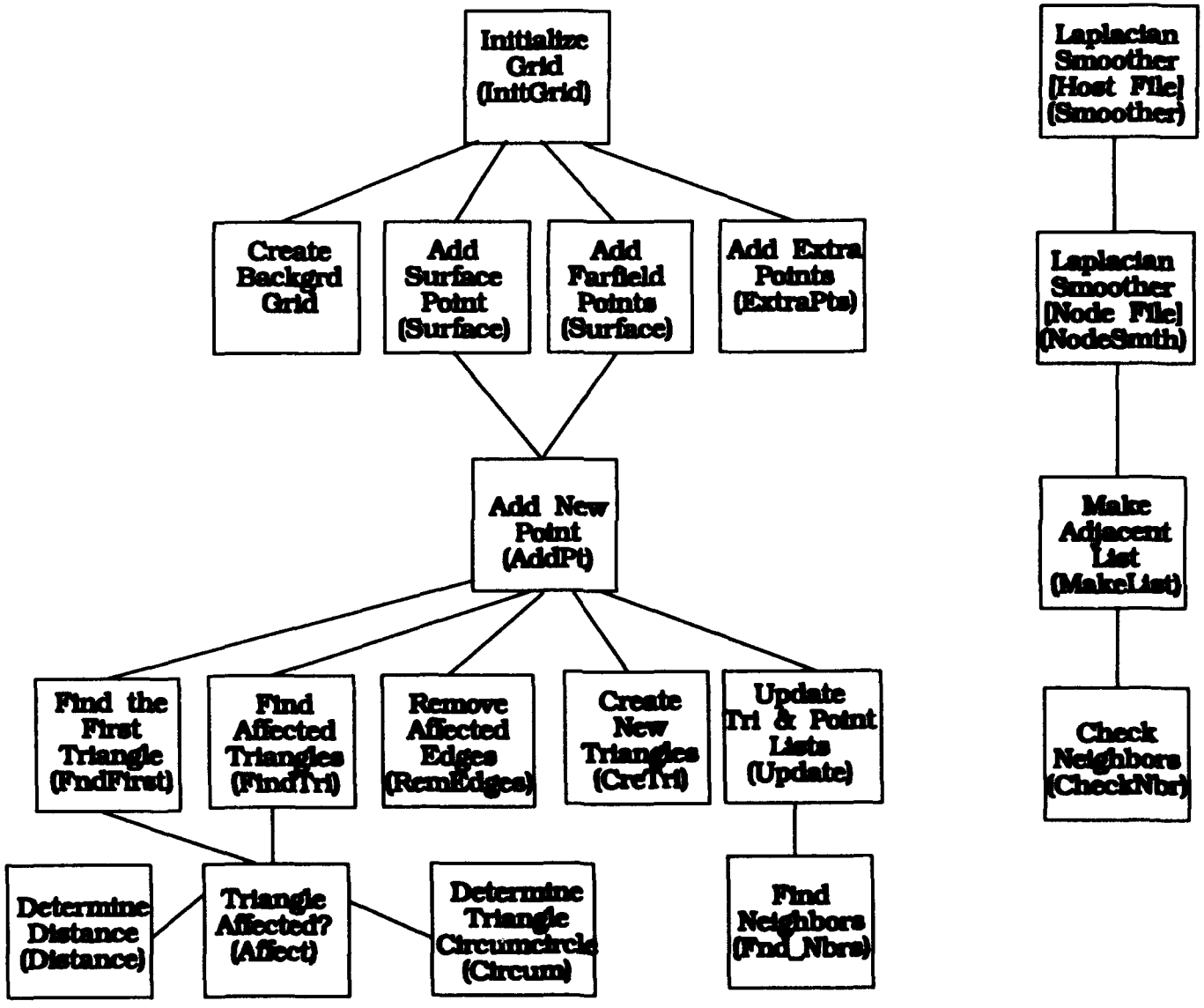


Figure 3-4. Grid Generator Software Structure Continued

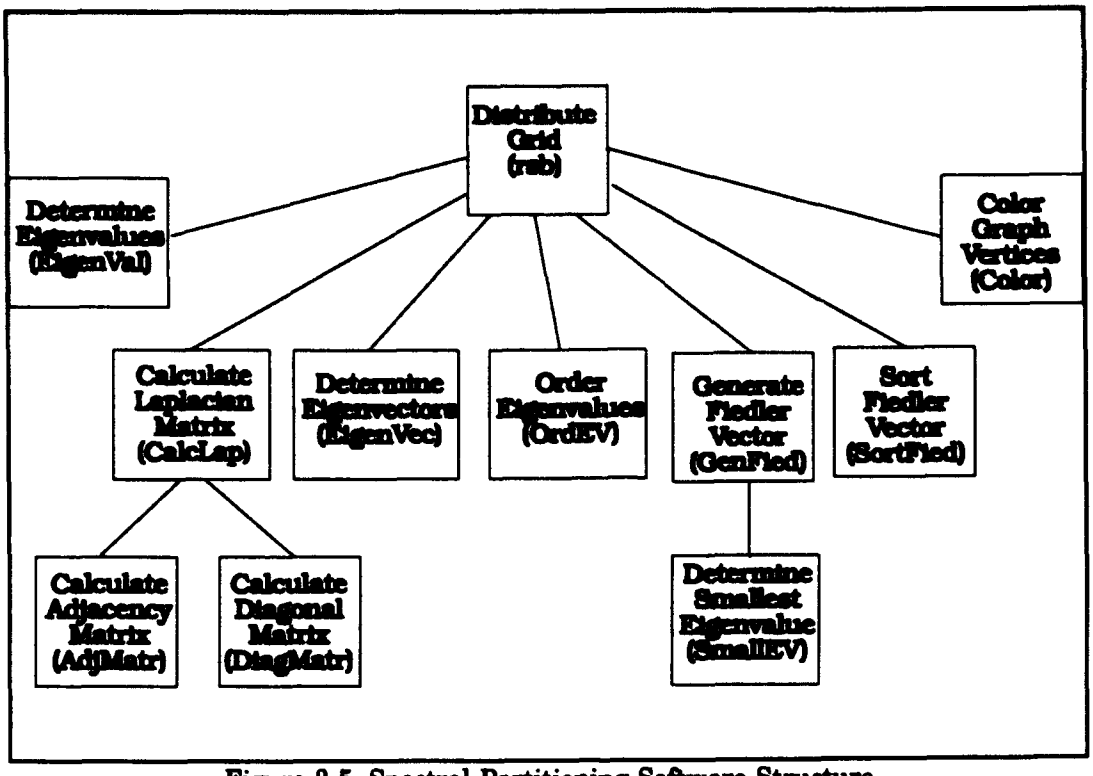


Figure 3-5. Spectral Partitioning Software Structure

Input. The input required by the software consists of two ASCII files. The first file consists of point coordinates for the surface of the object. The second file consists of the radius and point coordinates for the far-field boundary. These files are created through the NewInit portion of the software (see Figure 3-3). When the software is started, it prompts the user for the names of the files. It then asks if these are new files to be created. If the files are new, the software continues to prompt for the far-field radius and the number of far-field points, and the surface description. For sample program sessions, see Appendix A. The surface can be an ellipse or a cylinder. When an ellipse is generated, the number of points on the surface is twice the input number minus two, one on the leading edge and one on the trailing edge.

The input files have specific formats. The first line of the surface input file consists of the number of points in the file represented as an integer. The remaining lines consist of the x and y coordinates of the points making up the surface file. The coordinates are

represented as double precision floating point numbers. The far-field input file is similar, with the exception of the first line, which consists of the far-field radius, represented to double precision. The remaining lines follow as described for the surface file.

Output. The output generated by the software is an ASCII input file for Tecplot software. The Tecplot software is hosted on a Sun workstation which is used by CFD researchers at AFIT (Tecplot, 1992). The file contains a list of point coordinates followed by a list of triangles consisting of the three points which make up the triangle. From this information Tecplot creates a visual representation of the grid. To use a Tecplot input file, a user must run the ASCII file through the preprocessor, *preplot*, before using the main program, Tecplot, to display the grid. The grid can be displayed on the workstation screen or sent to a printer or file. GridGen prompts the user for an output file name and a title string to be included in the Tecplot file.

The Tecplot file has a specific format that must be followed. The first three lines are as follows, where the information in italics is application specific:

TITLE = "*grid title*"

VARIABLES = X, Y

ZONE I = *number of points* J = *number of triangles* F = FEPOINT

The next portion of the file contains the x and y coordinates of each point in the grid. Finally, all the triangles are listed, one per line, as a list of four integers. The first three are the points making up the triangle and the fourth is a repeat of the last point. This fourth "point" is required because Tecplot deals with quadrilateral rather than triangular shapes.

Code Development. The code is written for compilation by a Green Hills Fortran-386 compiler hosted on an Intel iPSC/2 with 80386 processors (Fortran Compiler, 1987). This compiler implements the ANSI Fortran-77 standard. Both versions of the grid generator consist of thirty-three subroutines contained in separate files. Two of the subroutines are

different between versions. This count includes one main host program, GridGen, and two node programs, NodeGen and NodeSmth.

Only portions of the code are parallelized. The boundary definition, initial grid creation, and 360-degree distribution software modules are not parallelized. The boundary definition and initial grid creation modules would be difficult to parallelize because the points making up the surface and far field boundaries are contained in two files. Writing and reading these files in parallel would present significant bottlenecks. The 360-degree distribution software is not parallelized because it is run only once when there are few grid triangles. Communication would dominate any attempt to parallelize this portion of the code. Portions of the RSB code are parallelized and are discussed later under the sub-section titled RSB. The point creation and addition portion of the software is parallelized. This portion includes the clustering and smoothing routines. These portions of the software are computationally intensive and are thus, good candidates for parallelization.

Version 1. This version of GridGen includes two subroutines, Exchange and ExtraPts, which are not present in the other version. Because this version does not implement interprocessor communication during point insertion, the portion of the grid distributed to each processor is modified after each iteration by moving the boundaries half the total size of the section assigned to a processor. This provides a method for grouping the triangles differently to ensure a processor will eventually have all the triangles necessary to complete the addition of a point. The Exchange subroutine performs this function.

The ExtraPts subroutine inserts additional points into the initial grid to allow for parallel point insertion. This subroutine was developed to increase the scalability of the software without introducing interprocessor communications during point addition. The routine inserts points along the section boundaries that define the grid partitions. The routine can insert up to ten additional points along each boundary. Beginning with the far-field boundary points, the new points are spaced such that a point is 3/4 of the distance between the surface and the previous point. This distribution pattern mimics the pattern

of a grid generated without additional points. New points are inserted on each boundary only until the generation of the grid is scalable. A sample initial grid without extra points is shown in Figure 3-6. In this example, only three points were required on each boundary before the grid generation would scale to two processors. Correspondingly, an initial grid with extra boundary points is included in Figure 3-7. The results of the extra point additions are presented in Chapter 4.

Version 2. This version includes facilities for interprocessor communication during point insertion. These facilities eliminate the need for modification of the initial distribution after each iteration. This change required changes to the generation and clustering processes. There are two modules not present in Version 1 of the software: Updt_Tri and Chk_Aff. Updt_Tri updates the triangle information changed by the insertion of a point on another node. Chk_Aff checks to see if a given triangle is affected by the addition of a new point on another processor. After a node completes the examination of one triangle, including the possible insertion of a new point, it determines if another processor is requesting information on a triangle in its domain. If information is requested, the node provides the information and waits for any changes in its domain before continuing to its next triangle. Although the wait for changes increases the processing time, not waiting could result in the simultaneous modification of a triangle by multiple nodes. An array indicating the processor to which every triangle is assigned is included on every processor. The decision to distribute the array rather than maintain it only on the host was made to help eliminate the message passing overhead that would have been involved with the other option.

Another possible problem is deadlock. If two or more processors are waiting for information from each other, they would wait forever without outside interruption. There are four main forms of deadlock, with variations on each. The four main forms are shown graphically in Figure 3-8. Although deadlock can occur between any number of nodes, the examples use four nodes. Form *A* in Figure 3-8 represents "simple" deadlock. The host

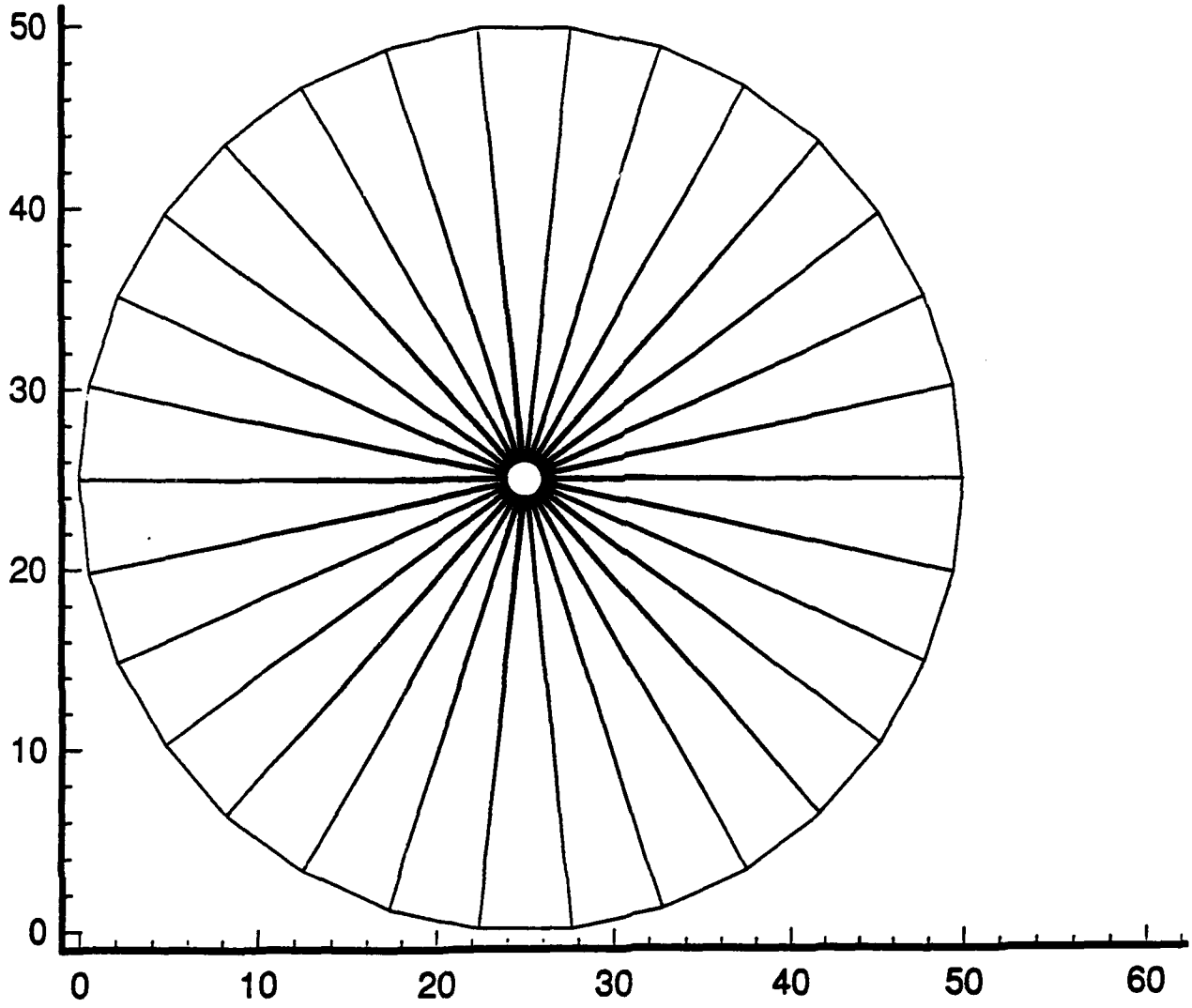


Figure 3-6. Initial Grid without Clustering or Smoothing

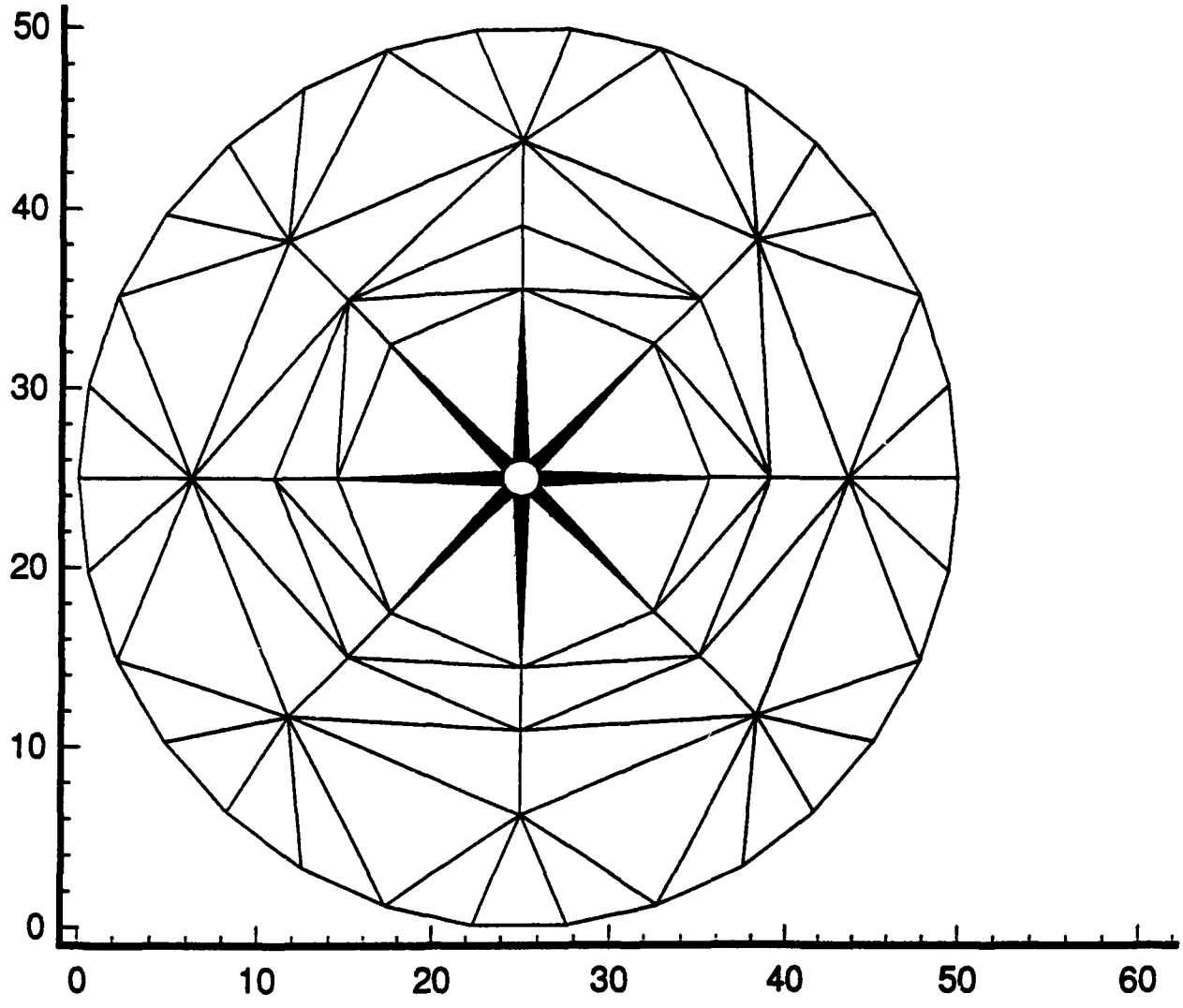


Figure 3-7. Initial Grid with Extra Points

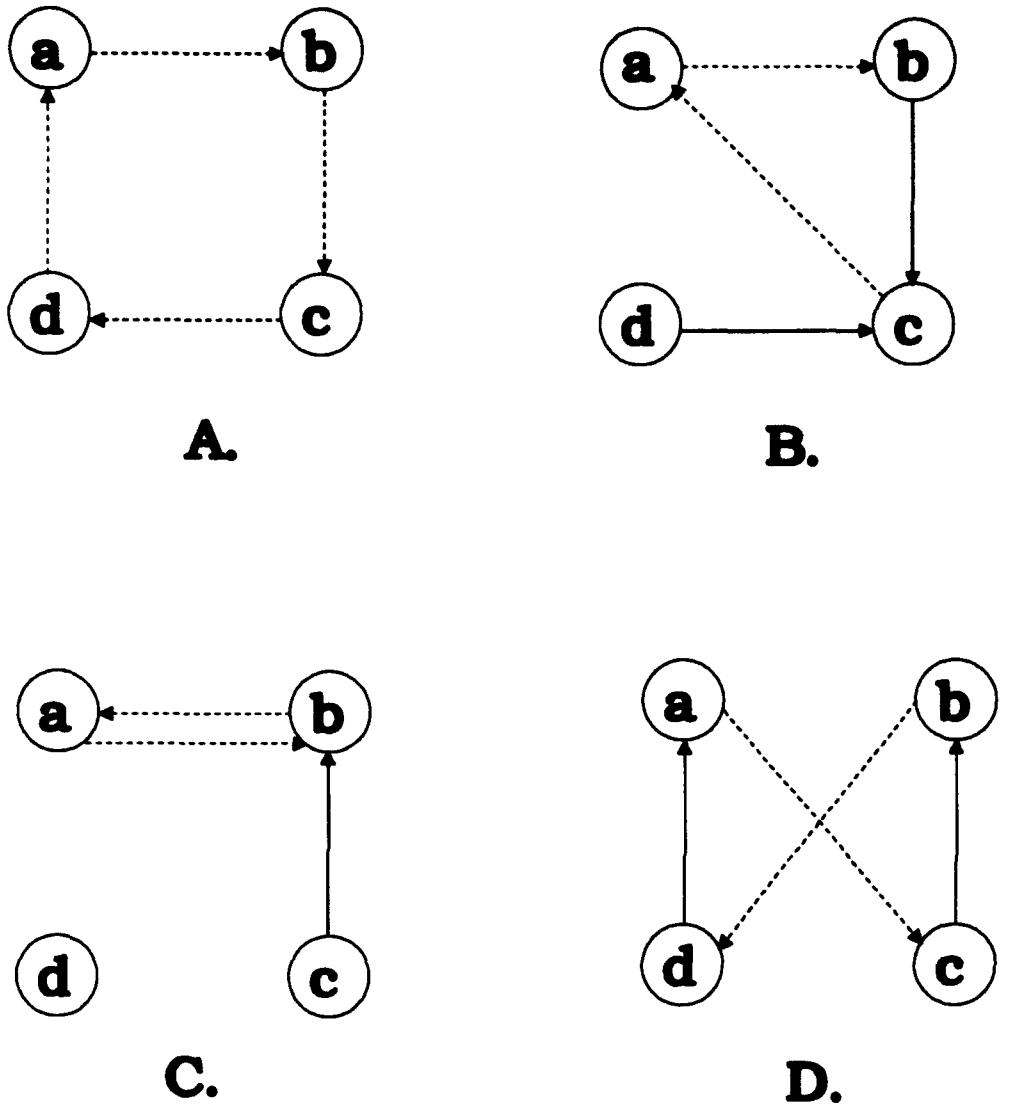


Figure 3-8. Possible Forms of Deadlock

program determines if simple deadlock exists by tracking the processors notifying it of possible deadlock. If deadlock exists, the host program causes one node to interrupt its normal processing and provide information to a requesting node. The host interrupts the processor with the lowest node number. Since this situation is not likely to occur often, a completely "fair" interruption algorithm is not required. This interruption results in restarting the examination of the current triangle following the servicing of the waiting node(s).

Form *B* represents a similar situation, except that node b is synchronized with node c and will not terminate until node c terminates. In this case, the message request from node a must be cancelled to allow it to answer the request from node c. To accomplish this, node c requests information from the host on the deadlock status of node a. If node a is waiting for a node synchronized with node c, then node c requests that the host terminate node a's request. This has the same result as described for form *a* above.

Form *C* in Figure 3-8 shows a variation on form *A*. In this case, either node a or node b will lose their message request. If node a's request is cancelled, no further cancellation is required. If node b's request is cancelled, however, node b must also release node c which is synchronized with it. As described for form *A*, the node with the lowest node number will be interrupted and move on to service the other node's request.

Form *D* shows a situation in which two nodes are waiting for responses from two other nodes which are synchronized with the opposite active node. In this case, either node a or node b must terminate it's point addition and release the node synchronized with it. The software deals with this form of deadlock through a timer. When a node finds itself in this situation it waits a predefined amount of time and then "dies", releasing all the nodes synchronized with it. This releases the synchronized node and enables it to answer requests from the other requesting node. Since both nodes activate timers, the first to start the timer will be the one to "die".

RSB. The code examined for the RSB distribution method (see chapter 3 for a complete description of recursive spectral bisection) is a modified version of code obtained from researchers at Oak Ridge National Laboratory (Leete, Peyton, and Sincovec, 1993). The researchers indicated that the code is not exceedingly fast and is only the first step in the creation of an efficient RSB mapping tool (Peyton, 1993). The researchers are currently working on a multi-level RSB mapping tool which may be more efficient. The original code mapped grids based on distributing the points of the grid. The modified code distributes the grid based on the triangles of the grid. The RSB "implementation is based on a straightforward parallel implementation of the Lanczos method, with only the dot and matrix-vector products performed in parallel" (Leete, Peyton, and Sincovec, 1993: 924). The input required for the Oak Ridge code is a file in Harwell-Boeing format for sparse symmetric matrices. The Harwell-Boeing format consists of a four line header and up to four records consisting of column start pointers, row indices, numerical values, and the right-hand-side matrix (for a complete description of Harwell-Boeing format see the Harwell-Boeing description document) (Duff, Grimes, and Lewis , 1988). The RSB code only uses the row and column information to generate a connectivity graph associated with the matrix. The RSB distribution method was not incorporated into a version of GridGen because of the results obtained by the GridGen versions, as reported in Chapter 4.

Summary

This grid-generation algorithm is a combination of different methodologies, each appropriate at different stages of the algorithm. Watson's incremental insertion algorithm is used for point insertion. Holmes and Snyder's approach is used for point creation. The algorithm uses a discretized boundary definition. Clustering and smoothing functions presented by Anderson are used to improve the effect the grid has on the numerical solution of the flowfield. Two distribution methodologies are examined, 360-degree and recursive spectral bisection, although only the 360-degree methodology is implemented for

the GridGen software. The results achieved by this combined algorithm are presented in Chapter 4.

Chapter 4: Testing and Results

Introduction

This chapter presents the issues involved and the results of the testing of the CFD unstructured grid generator, GridGen. The testing section raises issues that are important in testing and focuses on how such testing was performed. The results are also included for each version of the parallel software. The testing issues involved in the analysis of the GridGen program include parameter modification, accuracy, timing analysis, scalability and load balancing. The test plan involved with each of these issues is discussed individually in this chapter under the appropriate headings.

Parameter Modification

There were a number of parameters that could be modified prior to grid generation. Modifying these parameters affected the accuracy of the final grid and the time required to generate the grid. The first two parameters were the maximum aspect ratio and area of the triangle allowed before a point is inserted at its circumcenter. The value of these parameters determined the size and shape possible for the triangles included in the final grid. The values for these parameters were hardcoded in the NodeGen program.

There are several parameters associated with the clustering function included in GridGen. These parameters are defined in the external file *Cluster.dat*. The first parameter, *do*, defines the distance from the surface where clustering mainly occurs. The second parameter indicates the number of iterations of clustering that the program should perform. The remaining values represent different values of β, *beta*, one for each iteration of the clustering function. The β parameters control how quickly the transition at *do* occurs. Normally, the values of β should increase slightly with each iteration. "Three or four repetitions in which β is gradually increased lead to grids with good clustering near

the surface of the airfoils, and a reasonably smooth transition region between the clustered and nonclustered areas is obtained" (Anderson, 1992: 6).

The simple introduction of the clustering function into the basic software had a significant effect on the number and location of points in the grid. The clustering function increased the proportion of triangles near an object's surface where the flow varies significantly and is more important to the determination of the flow field about the object. This change is shown in Figures 4-1 and 4-2 which were created by Tecplot. Figure 4-1 is a sample basic grid generated for a nearly cylindrical ellipse with a major axis of 1.0 and a minor axis of 0.9 on one node. Figure 4-2 shows the same grid after two iterations of clustering with a *do* of 0.5 and β values of 5.0 and 7.0.

The third set of parameters control the smoothing function. These parameters are included in the external file *Smoother.dat*. The first parameter indicates the number of smoothing iterations the program should execute. The second parameter, ω, is the relaxation factor. This parameter controls the size of the change in the x and y directions. Typical values of ω are near 0.2. One-hundred to two-hundred smoothing iterations are normally performed.

Once again, simply the introduction of the smoothing function had visible effects on the resulting grid. The Tecplot graphs in Figures 4-3 and 4-4 show how smoothing affects a grid created on two nodes for an ellipse with a major axis of 1.0 and a minor axis of 0.9. This example used a ω of 0.2 and fifty smoothing iterations. The software always performs clustering before smoothing.

The other parameters that can be modified, define the surface and far field boundaries of the grid. The far field boundary is defined by its radius. The distance from the surface to the far field boundary must be sufficient to include the entire area of interest. The far field boundary is considered to be in the free stream. The surface boundary is defined in GridGen as an ellipse or a circle. The size of an ellipse is defined by its major and minor

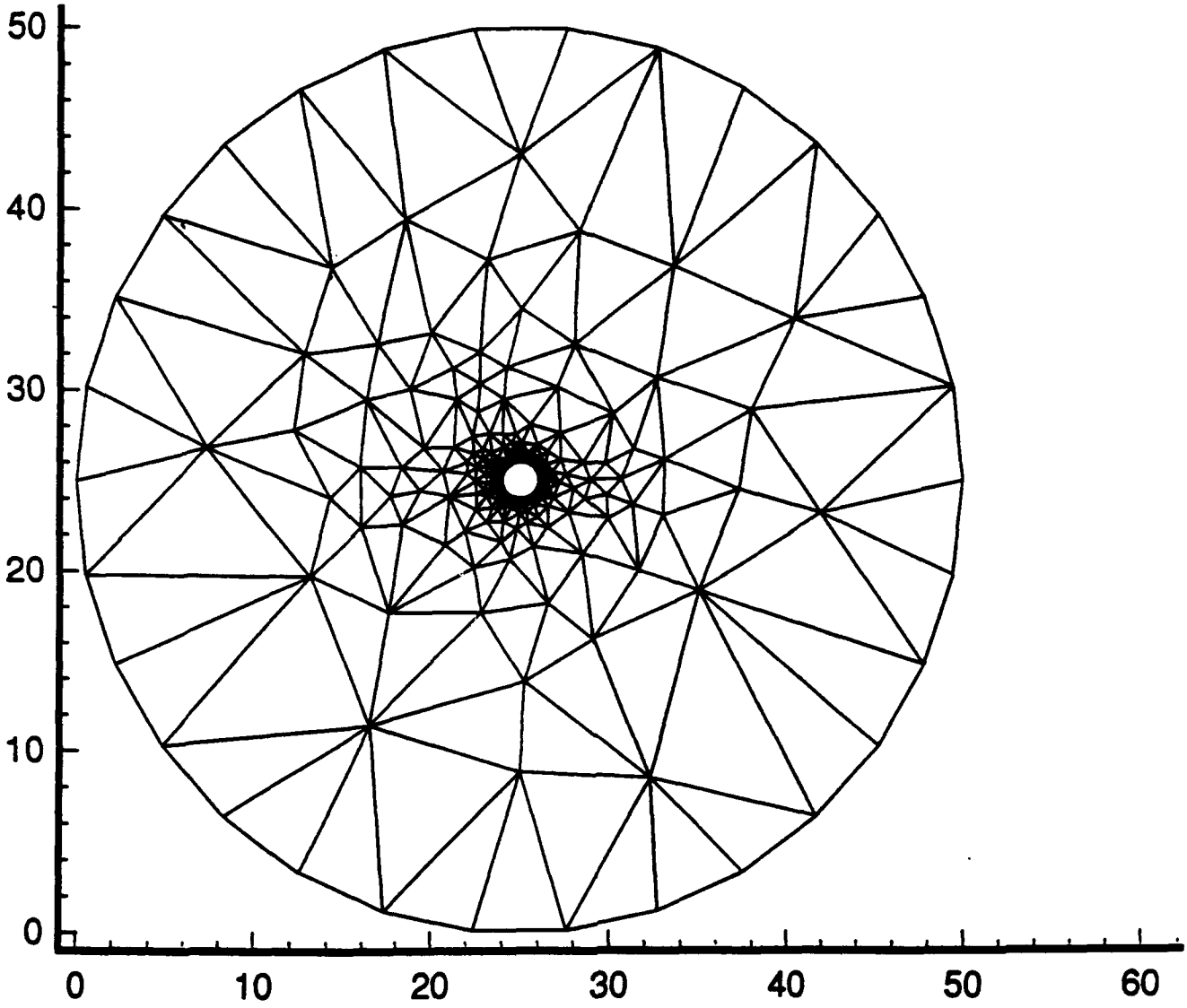


Figure 4-1. Grid with no Clustering and no Smoothing

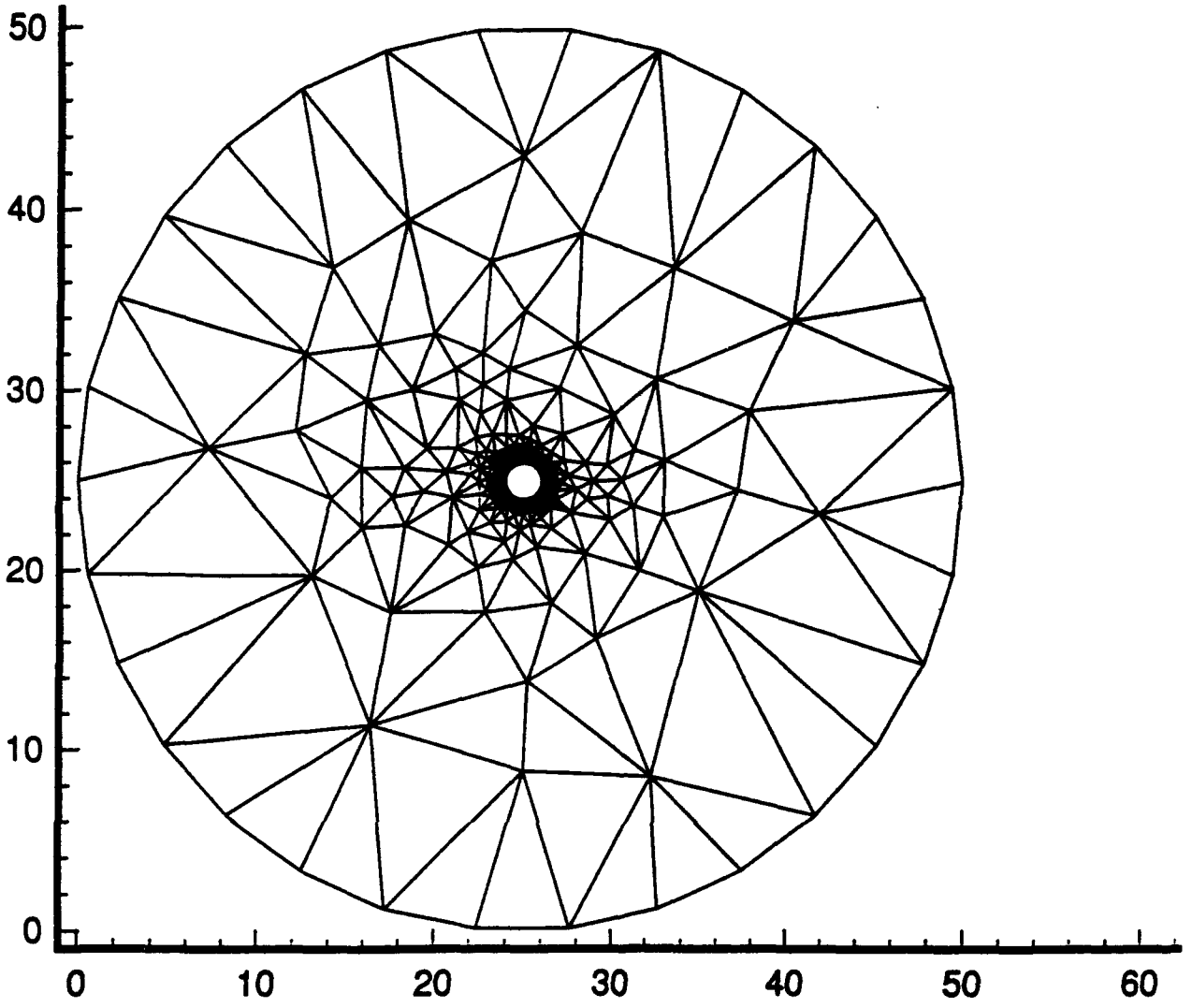


Figure 4-2. Grid with 2 Clustering Iterations and no Smoothing

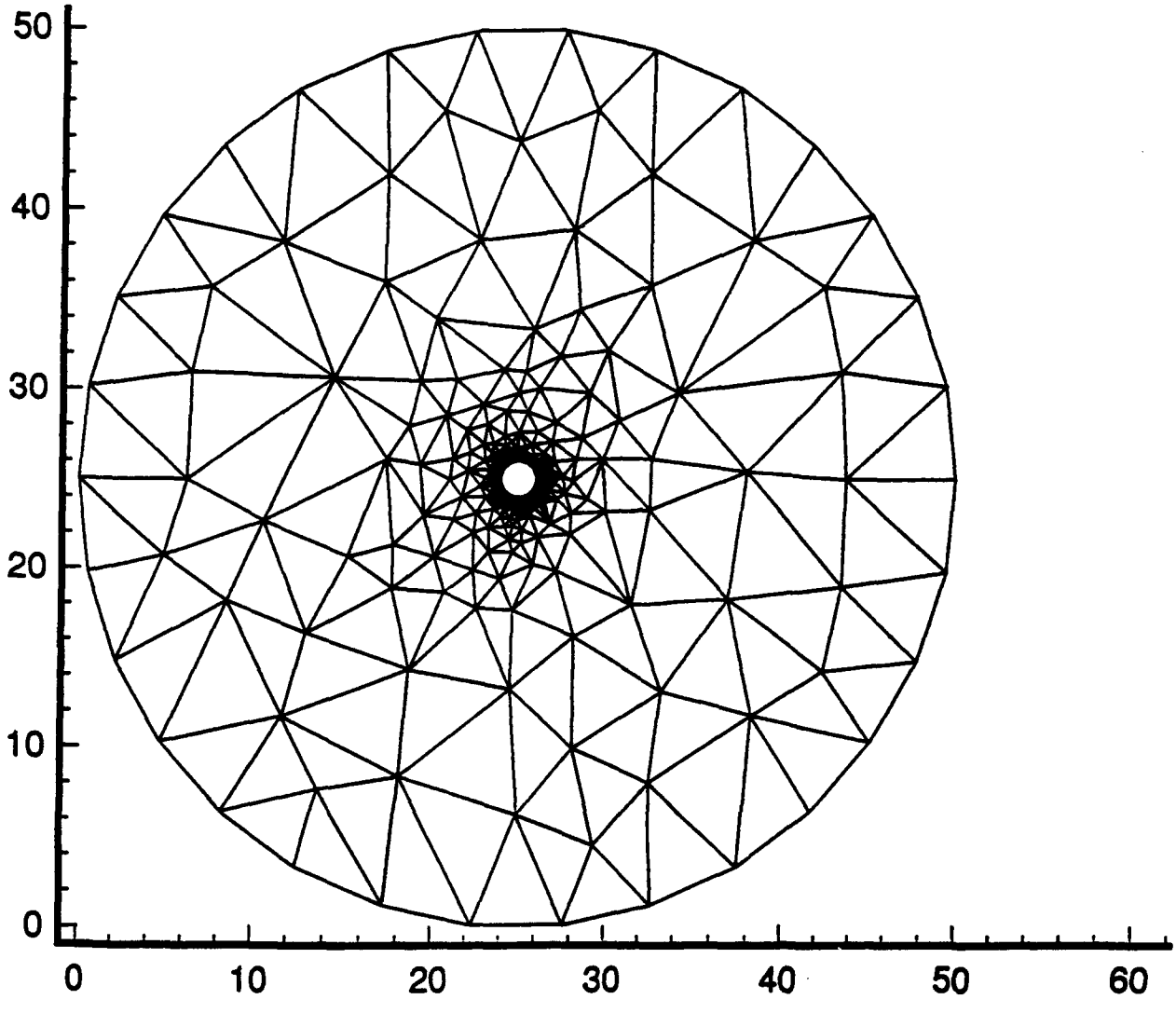


Figure 4-3. Grid with Clustering and no Smoothing

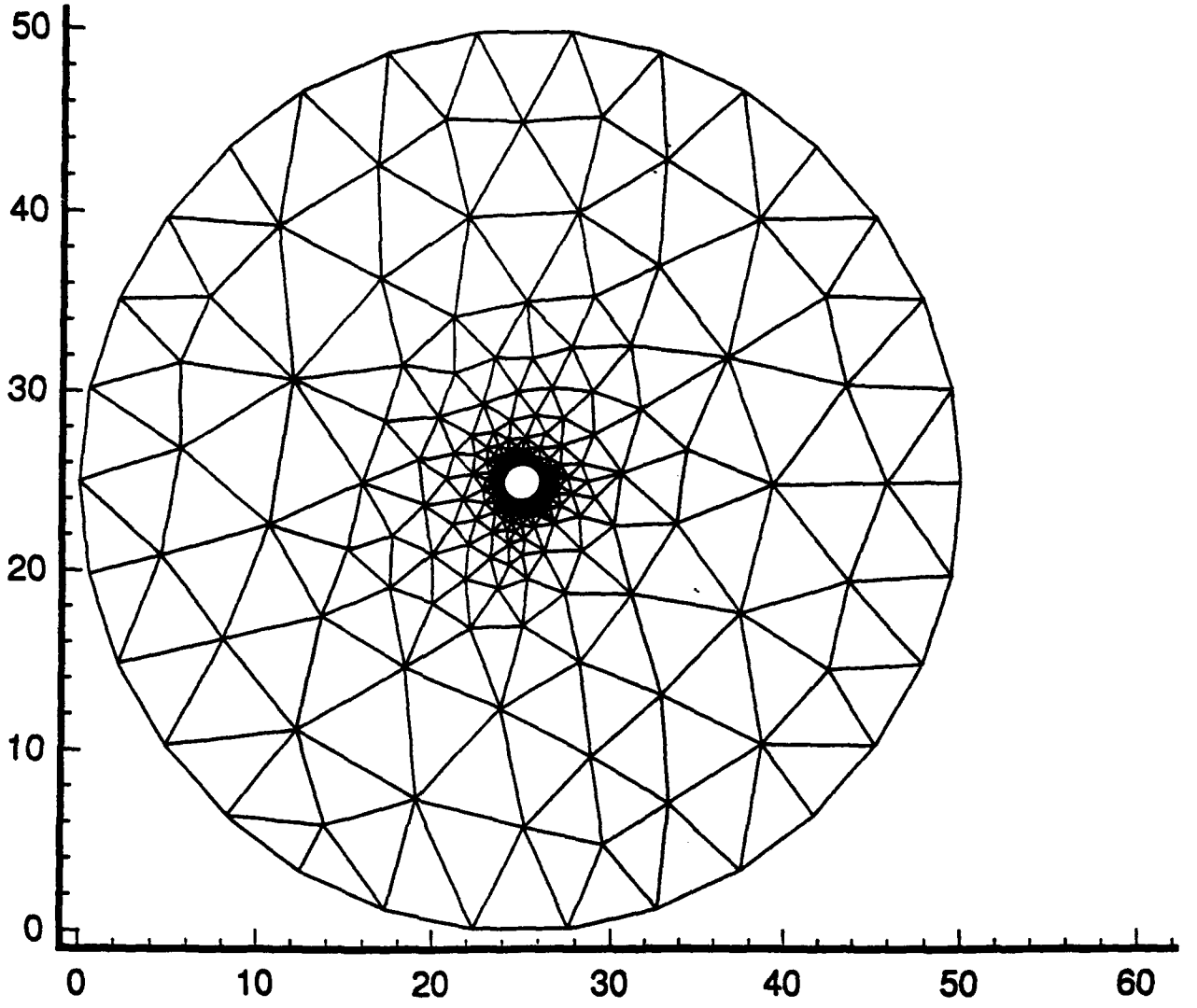


Figure 4-4. Grid with Clustering and 50 Smoothing Iterations

axes. The size of a circle is defined by its radius. The number of points included on the surface and far field boundaries is also parameterized. All of these parameters are set interactively when creating a new surface definition through GridGen (see Appendix A, Example 1).

Accuracy

The accuracy of the resulting grid depends on the parameters described earlier in this chapter. Balancing accuracy and speed of convergence during flow solution involves tradeoffs. Smaller, more regularly shaped triangles provide a more accurate flow field solution for a given problem, but the time required for such a solution may be exceedingly long. The inclusion of clustering and smoothing functions also improves the positive effect a given grid has on the accuracy of the numerical flow solver solution. The number of points and triangles in a resulting grid varied as the number of processors used varied. An example of this was seen on a grid generated about an elliptical surface which consisted of 610 points and 1214 triangles when generated on a single processor, but consisted of 632 points and 1258 triangles when generated on two processors using Version 1 of the software. This variation occurred because the order of point insertion varies as the number of processors changes.

The effect of the grid, generated by GridGen, on the accuracy was determined using a modified version of a sequential first-order finite-volume solver developed by Capt Frank Smith (Smith, 1992). The modifications included additional parameter passing (rather than common blocks), providing for grids with centers not located at coordinates (0.0, 0.0), and several parameter modifications due to the differences in implementation. Tests were run for an ellipse with a major axis of 1.0 and a minor axis of 0.9, and a circle with a radius of 1.0. Both test grids included two levels of clustering with a do of 5.0 and β values of 0.25 and 0.75 and one-hundred smoothing iterations with a relaxation factor of 0.2. The maximum aspect ratio was 3.5 and the maximum area allowed was 25.0. Table

4-1 shows the accuracy of the numerical solution using various grids as generated by Version 2 on two nodes of the iPSC/2 is compared to the results from Capt Smith's sequential grid generator run on a Sparcstation. This table indicates the maximum error of the numerical solution of the flow solver using the grid generated by the corresponding

Table 4-1. Maximum Error - Accuracy Results.

	Ellipse 0.9 / 1.0	Circle 1.0
GridGen Ver 2	0.201343	0.776127
Capt Smith's Grid Program	0.026967	0.004016

software. The results, as shown, indicate that the parallel GridGen software provides a grid which results in a numerical solution less accurate than the grid generated by the sequential software. An exact comparison of the grids generated by the parallel and sequential cannot be accurately made, however, for two reasons. The first is the uncertainty of the flow solver software used to determine the maximum error. The correctness of the software is not guaranteed because of the difficulties encountered during the transition of the software modules involved. The second reason is the range and precision of the floating point numbers available in the hardware. The Sun Sparcstation implements a 128-bit double-precision number, while the iPSC/2 implements a 64-bit double-precision number.

Timing

The length of time required by GridGen to generate a given grid was determined through the use of the iPSC command *mclock*. The software provided timing results for

each step in the generation process: boundary generation, initialization, distribution, generation, smoothing, and output. These code portions correspond to the program units shown on the software design in Figure 3-3. The times were expressed in milliseconds based on the time the Unix system used on the execution of the program. These timing results were compared to each other. The timing varied based on the grid size, the number of processors, and the number of clustering and smoothing iterations. To compare timing results for a single parameter, all other parameters were kept equal. Speedup and efficiency metrics were calculated to determine the effectiveness of parallelizing the code.

Version 1. The average timing results of each section of the code for an ellipse with a major to minor axis ratio of 1.0 to 0.9 on version 1 of the software is shown in Table 4-2.

Table 4-2. Timing Results for Version 1 1.0/0.9 Ellipse (in milliseconds)

Code Portion	1 Processor	2 Processors
NewInit	80	55
InitGrid	25800	57260
Distrib	3975	5340
Generate	25000	13893
Smoother	54310	54216
Output	4645	4860
Total	113830	135717

There was no speedup for this example as a whole. The parallel version of the problem took longer than the sequential. By breaking the code into separate portions it is possible to see where the problems lie. The portions of the code which are parallelized, Generate

and Smoother, ran faster on two processors than on one. The smoother, however did not demonstrate significant speedup because only fifty smoothing passes were used for this problem. The effect of the number of passes on the speedup in this portion of the code is discussed earlier in this chapter. On two processors, a speedup of 1.8 and an efficiency of 0.9 was demonstrated on the generation portion of the code. The total time required for generation of the software was negatively influenced by the addition of the extra points required to scale the software effectively. As described earlier in this chapter, a significant timing problem was shown by the times required to initialize the grid. The remaining times were for portions of the code executed sequentially under either setup, so it is not surprising that these times were relatively close together.

Version 2. The average timing results of each section of the code for an ellipse with a major to minor axis ratio of 1.0 to 0.9 on version 2 of the software is shown in Table 4-3. As occurred on Version 1, a speedup of 1.8 was achieved on the generation portion of the software on two nodes. A total speedup of 1.3 was also realized on two nodes. Further speedup on additional nodes was not achieved. This lack of speedup was due to the communications requirements of the software. As the number of processors increases, more communication and synchronous operation is required to add a single point. This synchronization is occasionally the result of deadlock avoidance. The avoidance methods often result in the abandonment of useful work done by a processor. This work must then be reaccomplished. The InitGrid portion of Version 2 requires over twice the time required by Version 1 on one processor. This occurrence is a direct result of the added complexity of the code that both the sequential and parallel portions of the code share.

Scalability / Load Balancing

The scalability and load balancing features of the software are intertwined. A lack of load balancing eliminates the possibility of scaling beyond a certain point. The testing involved in this analysis included increasing the problem size and/or the number of

Table 4-3. Timing Results for Version 2 1.0/0.9 Ellipse (in milliseconds)

Code Portion	1 Processor	2 Processors	4 Processors
NewInit	45	35	70
InitGrid	63520	63280	62890
Distrib	3880	3925	4040
Generate	150615	84570	111000
Smoother	71330	64325	70855
Output	6070	5670	6065
Total	295515	221875	254960

processors used to generate the grid. Increasing the problem size was limited to increasing the number of surface and far field boundary points. On the iPSC/2 available at AFIT, the number of processors was limited to eight.

Version 1. Version 1 of GridGen was minimally scalable. The scalability varied depending on the shape of the surface and the number of extra points added to the initial grid. Cylindrical or nearly cylindrical surfaces scaled better than markedly elliptical surfaces. This occurred because of the increased number of triangles required at the trailing and leading edges of an ellipse. Even cylindrical surfaces, however, required additional points to be included in the initial grid. The extra points reduced the dependence on neighboring triangles located on separate processors. This occurred only because of the lack of interprocessor communication during point insertion. The program could only add new points when all the triangles affected by the new point and their neighbors were assigned to the same processor. When the addition of extra points allowed

the software to scale to two processors, there were several iterations during which no useful work was completed. This occurred because a specific set of triangles had to be grouped on the same processor before a point could be added at the circumcircle of the triangle that needed to be changed.

The results of the addition of extra points were varied. Grid generation for cylindrical and nearly cylindrical surfaces scaled to two processors with the addition of extra points. The addition of points at only the two initial boundaries (straight up and straight down) was not successful. This type of addition actually made the problem worse because it increased the interdependence of triangles. The addition of points overcame the problem only when eight boundaries (every 45 degrees) on two processors were used. The number of points required at each boundary depended on the shape of the surface. Sometimes, a single point served to decompose the grid sufficiently, so only one point per boundary was added. Decidedly elliptical surfaces did not scale to two processors, even when additional points were added. This lack of scalability was a result of the interdependence of triangles on the leading and trailing edges of the surface which are more concentrated for an elliptical object than for a cylindrical one.

The effects of the addition of extra points on the timing of the grid generation algorithm was demonstrated by the timing differences seen in the initial grid creation portion of the code. In one test case, the only modification was the addition of a second processor. The main difference in the time the grid generation required was during the creation of the initial grid. On one processor, the initial grid generation required an average of 25800 milliseconds. On two processors, twenty-four additional points were added on the boundaries. The time required for initial grid generation averaged 57260 milliseconds.

Version 2. Although this version was capable of running on any number of nodes, the improvement gained was limited to two nodes. This limit was caused by the interdependence of the triangles making up the initial grid. The interdependence caused a

large number of messages to be exchanged between nodes, and communication time then dominated computation time.

Chapter 5: Conclusions and Recommendations

Introduction

This chapter presents the conclusions and recommendations resulting from this thesis research. The conclusions section discusses the results of the tests described in Chapter 4. This discussion includes the level of success achieved by the complete implementation and the differences between the successes of the different versions of the software. The recommendations section includes recommendations for GridGen code improvements and continued research in the area of parallel grid generators.

Conclusions

Both versions achieved a speedup of 1.8, or an efficiency of 90%, for the generation portion of the software on two nodes. Neither version achieved further speedup. Version 1 ran almost twice as fast as version 2, but could not execute on multiple nodes for some cases because of the interdependence between triangles. To successfully parallelize the algorithm, message passing is required to allow nodes access to triangles not assigned to them. The addition of message passing in version 2 required additional coding for deadlock detection and avoidance. This additional coding caused run times to decrease, even when the code was executing on only one node.

The desired speedup was not achieved by this implementation of this algorithm as applied to simple CFD surfaces on the Intel iPSC/2. The initial grid consisted of a small number of triangles stretching from the surface to the boundary. New points were added at the circumcenter of the triangles. The shape of the triangles caused many triangles to be affected when a new point was added because the new point was included in the circumcircles of many other triangles. The processors containing all affected triangles required synchronization. This synchronization limited the amount of time the processors could perform useful work in parallel.

Although both versions demonstrated an efficiency of 0.9 for the generation portion of the software, well above the 0.7 mentioned in Chapter 1, the comparisons are for parallel code running on one and two processors. A more accurate comparison would be the parallel code on two processors and sequential code on a single processor. It can be concluded that a sequential version of the grid generator could execute faster than the parallel code, because only Version 2 of the parallel code operates consistently on multiple nodes, and Version 1 executes twice as fast on a single processor, although the code is written for parallel execution,.

Recommendations

Code Improvements. There are several possible code improvements that may have a favorable effect on the efficiency of the code. There are three types of changes that could be made. The first type involves code changes that would save space and/or time. The second type includes changes that would improve the readability and maintainability. The final type of change is more complicated. It involves changing the approach to coding a portion of the code.

There are several examples of the first type of change. Throughout the code, there are times when searching the triangle or point list is implemented. For modules where this occurs frequently, an array indicating the location of a triangle in the list may be appropriate. The location array indicates which processor the triangle is assigned to, but not where the triangle is located on a processor's triangle list. This is not a consideration for the code executing on the host because the location of a point or triangle on the host's list is the same as the point or triangle identification number. A related change is modifying the location array to use a smaller integer type (such as INTEGER*2). This would save space on the processors. This is possible because only the number of processors need to be represented. Another, related, change involves the elimination of passing messages to the host indicating the location of new or modified triangles. This change is

possible because the host does not use the information after the initial distribution of nodes.

A change that may result in improved speedup for Version 2 of the software is to include the addition of extra points as was done in Version 1. The results, however, may be the same as occurred on Version 1 where the InitGrid portion of the code slowed the total time to the extent that the code ran faster on one node than on two.

There are several changes that would increase the readability and maintainability of the code. The first of these changes involves moving the code involved with taking care of the message requests of other nodes from the main modules and into a separate subroutine. This block of code is included in two places in the NodeGen program and in the cluster subroutine. The second change is relatively minor. The addition of EXTERNAL lines in each module would indicate to future users which subroutines are external. Another change that would increase the usability and maintainability of the software would be to modify the code to read the maximum aspect ratio and area values from a file. This change would provide easier access to these parameters and eliminate the need to modify the code to change the parameters.

The final type of change involves looking for solutions to problems that exist in the execution of the program. As the number of nodes used by the program increases, the number of deadlock situations also increases. This leads to a greater number of point additions that are cancelled and must start over at a later time. One possible solution would be to alter the code location where each node begins execution. A delay could also be included to prevent a node from moving on to its next triangle immediately if there are no outstanding messages. A careful balance must be struck to eliminate repetitive code execution and avoid excessive delays.

Another example of the final type of change involves attempting to modify the order in which the triangles are examined to increase continued load balancing. If all processors began triangle examination on the same side of their boundaries, then the load balancing

may remain more even. Currently, if two processors begin triangle examination near the same boundary, one processor dominates the boundary and gains more new triangles than the other, thus altering the load balance.

If this software is used for grid generation in the future, the accuracy of the numerical solution possible for grids generated by this software must be validated. A separate interface could be written to move the output of the grid generator on the iPSC/2 to the Sparcstation for solving the flow field. The range and precision limits introduced by the hardware must still be considered.

Future Research. Further research in the area of parallel CFD grid generation using Delaunay triangulation should focus on other methods of point creation and insertion. To be successful on simple surfaces, researchers will need to determine a way to decouple the interdependence of the triangles in the initial grid. A possible avenue of research is to use a divide and conquer method for point generation. This type of algorithm would allow processors to perform Delaunay triangulation on separate portions of a grid and then define the areas between the separate areas. The methods used in this research may be more applicable for complex grids in which the interdependency of the triangular elements is not as pronounced as it is for a simple circle or ellipse. Other types of flows, such as internal flows, should also be examined. Adaptive gridding should be examined once a successful parallel implementation of a single-pass grid generator is complete.

Researchers have shown that recursive spectral bisection is a successful domain decomposition technique for CFD applications. For the initial grid generation process, RSB may be especially costly in terms of time. Because of the long triangle sides in an initial grid, the "cuts" made by a RSB distribution method will be similar to the simple 360-degree methodology used in the GridGen software. RSB is much more likely to be useful after the initial grid generation is complete and there are a large number of triangles and ways to distribute the triangles amongst processors. A parallel version of RSB is also an important topic for continued research because of the time required to distribute a grid using RSB.

Summary

This research demonstrated one possible combination of methodologies that was not successful in achieving continued speedup on parallel processors, specifically the Intel iPSC/2. Further research is required to determine if using Delaunay triangulation, rather than advancing front, in conjunction with parallel computing is a valuable tool for unstructured grid generation for CFD applications.

Appendix A: Sample Program Sessions

This is a sample run on one node for an ellipse object, including the creation of the input files.

```
c386 94:gridgen
 Do you want to generate new files?
y
 Enter Surface file name
surface.dat
 Enter Far Field file name
farf.dat
 Select Body Type (enter corresponding number)      -- define surface
   1) Elliptical Cylinder
   2) Circular Cylinder
1

 Enter the radius of the far field boundary:
25
 Enter the number of far field points:
30
 Enter the number of surface points (on each side):
50
 Enter the semi-major axis length:
1.0
 Enter the semi-minor axis length:
.9

 Distributing Grid

 Generating Grid

          0          132          258          -- Grid Generation
          0          401          796          --   Iterations
          0          480          954          -- Node ID, Num Pts,
          0          491          976          --   Num Tris

 Clustering Triangles

 Smoothing Grid


 Enter the name of the output file:
test1.out
 Enter the title for the file:
```

'Ellipse 0.9 / 1.0 on 1 Node'

```
Number of Points:       777
Number of Triangles:   1548


Start Time      350
NewInit         560
InitGrid      63770
Distrib        4200
Generate     149760
Smoother      72030
Output         6090
Total Time   296460
```

This is a sample run for the same ellipse on two nodes without creating the input files.

```
c386 97:gridgen
 Do you want to generate new files?
n
 Enter Surface file name
surface.dat
 Enter Far Field file name
farf.dat

 Distributing Grid

 Generating Grid

          0          70         129
          1          70         129
          0         195         353
          1         220         405
          0         232         424
          1         272         506
          1         276         513
          0         237         433

 Clustering Triangles

 Smoothing Grid


 Enter the name of the output file:
test2.out
 Enter the title for the file:
```

'Ellipse 0.9 / 1.0 on 2 Nodes'

Number of Points: 728
Number of Triangles: 1450

Start Time	370
NewInit	30
InitGrid	63550
Distrib	3920
Generate	84720
Smoother	63980
Output	5650
Total Time	221880

Bibliography

Agarwal, Ramesh K. "Development of a Navier-Stokes Code on a Connection Machine." *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications.* 917-924. March 1989.

Anderson, Kyle W. *Grid Generation and Flow Solution Method for Euler Equations on Unstructured Grids.* NASA Technical Memorandum 4295. NASA, April 1992.

Angus, I.G. and W.T. Thompkins. "Data Storage, Concurrency, and Portability: An Object Oriented Approach to Fluid Mechanics," *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications.* 925-929. March 1989.

Arthur, Trey. and Michael J. Bockelie. *A Comparison of Using APPL and PVM for a Parallel Implementation of an Unstructured Grid Generation Program.* NASA Contractor Report 191425. NASA, January 1993.

Barszcz, Eric, Tony F. Chan, Dennis C. Jespersen, and Raymond S. Tuminaro. "Performance of an Euler Code on Hypercubes," *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications.* 933- 940. March 1989.

Barth, Timothy J. "Aspects of Unstructured Grids and Finite-Volume Solvers for the Euler and Navier-Stokes Equations," *AGARD Special Course on Unstructured Grid Methods for Advection Dominated Flows.* AGARD Report 787: 6-1:61, May 1992.

Beran, Philip S. "Analysis And Overview Of CFD Algorithms," presented at the *Computational Fluid Dynamics Short Course.* Air Force Institute of Technology, Wright-Patterson AFB OH, September 1992.

Braaten, Mark E. "Computational Fluid Dynamics on Hypercube Parallel Computers," *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications.* 949-951. March 1989.

Braaten, Mark E. "Parallel Computation of the Compressible Navier-Stokes Equations with a Pressure-Correction Algorithm," *Proceedings of the 5th Distributed Memory Computing Conference.* 463-467. Charleston SC: IEEE Computer Society Press, April 1990.

Committee on Physical, Mathematical, and Engineering Sciences. *Grand Challenges: High Performance Computing and Communications.* National Science Foundation. 1991.

DeCegama, Angel L. *Parallel Processing Architectures and VLSI Hardware.* Englewood Cliffs NJ: Prentice-Hall, Inc., 1989.

Duff, Iain S., Roger G. Grimes, and John G. Lewis. *User's Guide for the Harwell-Boeing Sparse Matrix Collection.* Harwell Laboratory, Oxon England and Boeing Computer Services Seattle WA, 1988.

Fortran Compiler. Green Hills Software, Inc. 1987.

Gane, Chris and Trish Sarson. *Structured Systems Analysis: tools and techniques*. New York NY: Improved System Technologies, Inc., 1979.

George, P.L. *Automatic Mesh Generation*. Chichester England: John Wiley & Sons, 1991.

Hauser, Jochem and Roy Williams. "Strategies For Parallelizing a Navier-Stokes Code on the Intel Touchstone Machines," *International Journal for Numerical Methods in Fluids, Vol 15*. 51-58. John Wiley & Sons, Ltd, 1992.

Hendrickson, Bruce, and Robert Leland. "An Improved Spectral Load Balancing Method," *Proceeding of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, Vol II*. 953-961. Philadelphia: SIAM, 1993.

Holey, J. Andrew and Oscar H. Ibarra. "Triangulation, Veronoi Diagram, and Convex Hull in k-Space on Mesh-Connected Arrays and Hypercubes," *1991 International Conference on Parallel Processing Vol III*. 147-150. 1991.

Intel Corporation. *iPSC/2 Fortran Programmer's Reference Manual*. Beaverton OR: Intel Scientific Computers, 1988.

Leete, Charles A., Barry W. Peyton, and Richard F. Sincovec. "Toward a Parallel Recursive Spectral Bisection Mapping Tool," *Proceeding of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, Vol II*. 923-928. Philadelphia: SIAM, 1993.

Lewis, Ted G. and Hesham El-Rewini. *Introduction to Parallel Computing*. Englewood Cliffs NJ: Prentice-Hall, Inc., 1992.

Löhner, Rainald, Jose Camberos, and Marshall Merriam. "Parallel Unstructured Grid Generation," in *Unstructured Scientific Computation on Scalable Multiprocessors*. Eds. Piyush Mehrotra, Joel Saltz, and Robert Voigt. Cambridge MA: MIT Press, 1992.

Long, Lyle N., M.M.S. Khan, and H. Thomas Sharp. "Massively Parallel Three-Dimensional Euler/Navier-Stokes Method," *AIAA Journal, Vol 29, No 5*. 657-666. (May 1991).

Mavriplis, D.J., Raja Das, Joel Saltz, and R.E. Vermeland. "Implementation of a Parallel Unstructured Euler Solver on Shared and Distributed Memory Architectures," *Supercomputing '92 Proceedings*. 132-140. IEEE Computer Society Press, Nov. 1992.

Olander, Daryl, and Robert B. Schnabel. "Preliminary Experience in Developing a Parallel Thin-Layer Navier Stokes Code and Implications for Parallel Language Design," *Proceedings of the Scalable High Performance Computing Conference (SHPCC)*. 276-283. IEEE Computer Society Press, April 1992.

Peyton, Barry W. "RSB Code." Electronic Message. 30 September 1993.

Scherr, S.J. "Implementation of an Explicit Navier-Stokes Algorithm on a Distributed Memory Parallel Computer," *31st Aerospace Sciences Meeting and Exhibit*. 1-9. AIAA 93-0063. January 1993.

Smith, Frank. *An Unstructured 2-Dimensional Grid Generator*. AERO 899 Project. Wright-Patterson AFB OH; Air Force Institute of Technology, 1992.

Stagg, A.K., and G.F. Carey. "Massively Parallel MIMD Solution of the Parabolized Navier-Stokes Equations," *Proceedings of the Scalable High Performance Computing Conference SHPCC-92*. 328-335. IEEE Computer Society Press: April 1992.

Tecplot Version 5. Amtec Engineering, Inc., Bellevue WA: January 1992.

Vita

Captain Deborah E. Davis was born on 8 November 1965 in Gold Beach, Oregon. She graduated from Big Sky High School in Missoula, Montana in 1984. She attended the U.S. Air Force Academy, graduating with a Bachelor of Science in Computer Science on June 1, 1988. Upon graduation she received a regular commission in the USAF and was assigned to the Defense Intelligence Agency located at Bolling AFB in Washington D.C. She was a Data Communications System Programmer responsible for providing world-wide intelligence analysts with reliable computer services on both IBM and DEC systems. Her job duties included installation, troubleshooting, and system administration. In May 1992 she was reassigned as a student in the Graduate School of Engineering at the Air Force Institute of Technology (AFIT) located at Wright-Patterson AFB, Ohio. Following graduation from AFIT, she will be assigned to U.S. Strategic Command at Offutt AFB near Omaha, Nebraska.

Permanent Address: 1185 Vicki Drive

Missoula, MT 59801

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	DEC 93	Master's Thesis

4. TITLE AND SUBTITLE
A PARALLEL COMPUTATIONAL FLUID DYNAMICS UNSTRUCTURED GRID GENERATOR

5. FUNDING NUMBERS

6. AUTHOR(S)
Capt Deborah E. Davis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology,
WPAFB OH 45433-6538

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/93D-05

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aerospace Plane - Joint Program Office
ASC/NAO
Wright-Patterson AFB, OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research addressed the development of a parallel computational fluid dynamics unstructured grid generator using Delaunay triangulation. The generator is applied to simple elliptical and cylindrical two-dimensional bodies. The methodologies used included Watson's point insertion algorithm, Holmes and Snyder's point creation algorithm, a discretized surface definition, Anderson's clustering function, and a Laplacian smoother. The first version of the software involved a processor boundary exchange at the end of each iteration with no inter-processor communications during the iterations. The second version used inter-processor communication during each iteration instead of the boundary exchange. Version 1 demonstrated a speedup of 1.8 for some portions of the code, but proved to be unscalable for more than two nodes due to the interdependency of the triangular elements. The results of Version 2 were similar. Two distribution methodologies, a simple 360-degree distribution and recursive spectral bisection (RSB), were examined. For the initial grid distribution, the distribution generated by the RSB code would be similar to the distribution generated by the 360-degree methodology and would require significantly more time to execute.

14. SUBJECT TERMS
Fluid Dynamics, Grid, Triangulation,
Parallel Processing, Computer Programs

15. NUMBER OF PAGES
86

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL